Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

JavaCO – Uma variante do Java com um sistema de tipos baseado em covariância

Duarte João Figueira Martins

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática

Orientador

Prof. Doutor Artur Miguel Andrade Vieira Dias

Lisboa

2010

Nº do aluno: 29757

Nome: Duarte João Figueira Martins

Título da dissertação:

JavaCO – Uma variante do Java com um sistema de tipos baseado em covariância

(JavaCO – A Java variant with a covariance-based type system)

Palavras-Chave:

linguagens de programação

sistema de tipos

covariância

compilação

Keywords:

programming languages

type system

covariance

compilation

Agradecimentos

Quero agradecer ao orientador, Professor Artur Miguel Dias, pelas ideias sugeridas, pela crença que depositou no meu trabalho acima de tudo pela ajuda e disponibilidade mostrada durante todo o processo de preparação e elaboração desta dissertação.

Quero também agradecer ao meu pai, João, à minha mãe, Maria Antónia e à minha avó, Maria, por todo o apoio que me deram ao longo da elaboração deste trabalho.

Também uma palavra de apreço aos meus amigos, pelo simples facto de serem meus amigos.

Obrigado a todos!

Resumo

Quando se compara a linguagem Eiffel com a linguagem Java, a diferença com mais impacto no estilo de programação é o facto de a primeira permitir métodos com argumentos covariantes, isto é, permitir a especialização do tipo dos argumentos à medida que se desce na hierarquia. Esta possibilidade é uma vantagem porque permite modelar determinados aspectos do mundo real de forma mais directa e intuitiva. Mas também há uma desvantagem que justifica a raridade desta opção na generalidade das linguagens modernas: o facto de criar complicações ao nível da tipificação estática dos programas.

O objectivo deste trabalho é criar uma variante do Java, chamada JavaCO, cujo sistema de tipos será baseado em covariância, à imagem do sistema de tipos da linguagem Eiffel. A introdução de um novo tipo genérico “This”, servirá para aumentar a utilidade prática do mecanismo de covariância. Espera-se assim proporcionar uma experiência de programação similar à da linguagem Eiffel, mas num ambiente de programação Java, usando as ferramentas e bibliotecas do Java.

A nova linguagem será implementada através da tradução para Java puro. A implementação seguirá as linhas da definição semântica e consistirá num tradutor escrito usando a ferramenta SableCC. O tradutor fará alguma análise estática dos programas de entrada; contudo, os aspectos mais essenciais da nova linguagem serão implementados por recurso a tipificação dinâmica (na mesma linha do Eiffel “clássico”).

Abstract

When comparing Eiffel language with Java language, the main difference about programming style is that the first one allows covariance in method's arguments, i.e. allows the use of specialized argument types as we go down in the hierarchy. This can be seen as an advantage as it allows you to model certain real world aspects in a more directly and intuitively way. But there is a disadvantage as well which justifies the rarity of this option in most of modern languages: the possibility of complications because of programs' static typing.

The main goal of this assignment is the creation of a Java variant, called JavaCO, with a covariance-based type system, mimicking the Eiffel type system. The introduction of a new generic type "This", will increase the practical usefulness of the covariance mechanism. It is expected to provide a programming experience similar to the Eiffel's one, however in a Java programming environment, using Java's tools and libraries.

The new language will be implemented by translation into pure Java. The implementation will follow the lines of the semantic definition and will result in a translator written using the tool SableCC. The translator will perform some static analysis of the input programs; however, the most essential aspects concerning the new language will be implemented using dynamic type checking (as in "classic" Eiffel).

Índice

1. Introdução 15

1.1. Motivação 16

1.2. Descrição e contexto 18

1.3. Solução apresentada e contribuições previstas 19

1.4. Estrutura da dissertação 20

2. Eiffel 23

2.1. Princípios de desenho 24

2.2. Desenho orientado a objectos 25

2.3. Desenho por Contracto 26

2.4. A arquitectura do software Eiffel 27

2.5. Entidades genéricas 30

2.6. Herança 31

2.7. Tipos e subtipos 34

2.8. Conformidade 35

2.9. Outros mecanismos e funcionalidades 36

2.10. Eiffel vs Java 37

3. Sistemas de Tipos 41

3.1. Para que servem os Sistemas de Tipos 42

3.2. Sistemas de tipos estáticos e dinâmicos 45

3.3. Relação de Subtipo 49

3.4. Covariância 51

4. Ferramentas 53
4.1. JavaCC 53
4.2. SableCC 54
4.3. Polyglot 58
4.4. Análise comparativa face aos objectivos do trabalho 59
5. Descrição do Trabalho 61
5.1. Objectivo do trabalho 61
5.2. Tradutor 62
5.3. Abordagem e princípios orientadores deste trabalho 63
5.4. Covariância nos argumentos dos métodos 65
5.5. Introdução da palavra-chave This 66
5.6. Uso estendido da palavra-chave This 66
5.7. Tipos genéricos e parâmetros de tipo 67
5.8. Implementação 67
6. Covariância 69
6.1. Covariância em JavaCO 69
6.2. Regra formal 70
6.3. Covariância noutras linguagens 71
6.4. Tradutor 72
6.5. Implementação 73
7. This (apenas nos parâmetros) 97
7.1. MyType 98
7.2. Métodos binários 99
7.3. Transformações associadas ao tipo This 100
7.4. Armazenamento de informação sobre parâmetros com tipo This 101

7.5. Validações relativas ao tipo This 102

8. This (também no resultado e variáveis) 103

8.1. Validação dos novos usos de This 104

8.2. Descrição detalhada do método adoptado 105

8.3. Implicações da palavra-chave final 105

8.4. Implementação 106

8.5. Considerações sobre o método adoptado 109

9. Genéricos e Parâmetros de tipo 111

9.1. Tipos genéricos e suas limitações 111

9.2. Parâmetros de tipo e suas limitações 113

9.3. Problema com cast 114

9.4. Particularidade do supply atendendo aos parâmetros de tipo 115

9.5. Representação interna dos parâmetros de tipo 116

9.6. Particularidade na substituição de tipos 116

9.7. Funcionalidades adicionadas 117

10. Trabalho futuro 119

10.1. Suporte de leitura de um conjunto de ficheiros em JavaCO 119

10.2. Possibilidade de herdar de classes JavaCO de biblioteca 120

10.3. Melhorar a eficiência do compilador 120

10.4. Indicação do número de linha no ficheiro original JavaCO de cada erro encontrado 121

10.5. Adição de outros mecanismos da linguagem Eiffel 121

11. Conclusões e análise crítica 123

Bibliografia 125

ANEXO I – código ilustrativo de alguns aspectos da tradução 127

1. Introdução

As linguagens de programação, apesar de um contínuo desenvolvimento ao longo das últimas décadas, não se podem considerar perfeitas. Entre aquelas que maior fama têm alcançado encontram-se as linguagens orientadas por objectos, pela intuição que o programador tem em colocar as suas ideias nos programas através da representação de dados em objectos, bem como na facilidade da sua aprendizagem. De entre essas linguagens destacamos o Java [Gosling, McGilton 96] e o Eiffel [Meyer 06], que estão intrinsecamente ligados a esta dissertação.

O Java é uma das linguagens de programação mais usadas nos dias de hoje. Facilmente nos apercebemos que o Java está em todo o lado: aplicações informáticas, jogos para telemóveis, aplicações wireless, aplicações para browsers Web, aplicações de comunicação remota, etc. No entanto, o Java tem algumas limitações que, apesar de não impedirem o seu uso a nível global, não permitem que o programador faça uso total da sua intuição aquando do desenvolvimento de software.

A linguagem Eiffel, desenvolvida por Bertrand Meyer, é também uma referência no mundo da programação. É usada nas mais diversas áreas, sendo a base do desenvolvimento de diversas aplicações financeiras, indústria e sistemas de telecomunicações.

Bertrand Meyer foi um dos principais proponentes da programação orientada por objectos, sendo que os seus ideais relativamente à programação passam pelo uso de linguagens simples, elegantes e amigáveis (“*user-friendly*”). Essa atitude foi crucial para o aparecimento e sucesso da linguagem Eiffel.

1.1. Motivação

A grande motivação desta dissertação passa por tentar aproximar um pouco mais a linguagem Java da linguagem Eiffel, isto é, acrescentar à linguagem Java alguns dos mecanismos do Eiffel que podem ser considerados uma mais valia, tendo em vista uma programação mais expressiva e intuitiva. Assim sendo, será desenhada e implementada uma variante da linguagem Java, o JavaCO, que terá um sistema de tipos mais semelhante ao que podemos encontrar em Eiffel, isto é, baseado em covariância [F. Miller et al. 10].

Este trabalho pretende também mostrar que a covariância pode ser útil em diversas situações em que o programador pretende expressar-se de forma clara, e que é possível integrar certos mecanismos interessantes do Eiffel na linguagem Java. Daí resultará a linguagem JavaCO.

E o que é a covariância? Covariância é a possibilidade de especializarmos os Tipos à medida que descemos numa hierarquia de classes. A linguagem Java já nos permite especializar o tipo de retorno de métodos, mas não os tipos dos argumentos. O exemplo seguinte irá ajudar na explicação daquilo que é a covariância, bem como da sua utilidade.

Java

```
class Animal {

        public void comer(Alimento al){
                ...
        }

        public void reprodução(Animal an){
                ...
        }

}

class Vaca extends Animal {

        public void comer(Alimento al) {
                ...
        }

        public void reprodução(Animal an) {
                ...
        }

}
```

Imagine-se que declaramos em Java uma classe *Animal*, e uma classe *Vaca* subtipo de *Animal*.

Foquemo-nos primeiramente no método *comer*, definido em *Animal* e redefinido em *Vaca*. Devido à invariância que existe em Java relativamente aos tipos dos argumentos de métodos, somos obrigados a manter o mesmo tipo na definição e todas as redefinições de um dado método: como se pode ver, em Java somos obrigados a manter o tipo *Alimento* no único argumento do método *comer*. Mas como sabemos, nem todos os animais ingerem o mesmo tipo de alimentos! É aqui que surge a vontade de especializar o tipo dos argumentos nas redefinições. Pretendemos expressar no nosso programa, por exemplo, que um animal do tipo *Vaca* só pode comer alimentos do tipo *Erva*, subtipo de *Alimento*.

Relativamente ao método *reprodução*, temos uma situação semelhante. Como sabemos, os animais reproduzem-se exclusivamente com animais da mesma espécie. Iremos também introduzir a covariância neste caso, através da introdução de um mecanismo novo e que será explicado mais à frente.

A necessidade de usar covariância surge muitas vezes em Java, em programas onde ocorrem duas ou mais hierarquias de classes ou onde seja necessário escrever métodos binários (métodos em que o tipo do argumento tem de ser igual ao tipo do receptor) – exactamente o caso do método *reprodução* do exemplo anterior. O programador de Java é tipicamente forçado a escrever código baseado em verificações dinâmicas de tipo, o que normalmente resulta em código de relativo baixo nível e de leitura por vezes difícil. Este trabalho responde a este problema fornecendo uma alternativa linguística que permite expressar directamente a covariância.

Em JavaCO, as intenções do programador podem ser expressas de forma directa através do seguinte programa:

JavaCO

```
class Animal {

        public covariant void comer(Alimento al){
                ...
        }

        public covariant void reprodução(This an){
                ...
        }

}

class Vaca extends Animal {

        public covariant void comer(Erva al) {
                ...
        }

        public covariant void reprodução(This an) {
                ...
        }

}
```

1.2. Descrição e contexto

O objectivo deste trabalho é a criação de uma nova linguagem, JavaCO, que consistirá numa variante da linguagem Java e que irá possuir um sistema de tipos ligeiramente diferente, baseado em alguns princípios que se podem encontrar na linguagem Eiffel. A tentativa de aproximação ao Eiffel consistirá na introdução de covariância nos tipos dos argumentos dos métodos.

Outro objectivo chave deste trabalho passa por assegurar a total compatibilidade com os programas em Java, significando isto que um programa em Java poderá ser visto como um programa em JavaCO, devendo manter a sua semântica.

A implementação será feita por tradução directa de JavaCO para Java, usando uma ferramenta de geração automática de reconhecedores sintácticos e de construtores de Árvores de Sintaxe Abstracta. A modelação em Java puro da covariância será feita recorrendo, em grande parte, a tipificação dinâmica.

Por fim, é de salientar que a implementação é focada na covariância, sendo que foi feito um esforço para deixar para o compilador de Java o tratamento de todos os outros aspectos, nomeadamente o tratamento de programas com erros de sintaxe contextual não relacionados com covariância.

1.3. Solução apresentada e contribuições previstas

Com vista à satisfação dos objectivos propostos, irá ser permitido o uso de argumentos covariantes na redefinição de métodos. Quer isto dizer que poderemos especializar o tipo dos argumentos de métodos herdados de outras classes. Um dos aspectos mais importantes passa por manter a invariância do tipo dos argumentos de métodos no código Java gerado. Assim sendo, e para respeitar as regras de tipificação do Java, no código resultante da tradução os métodos definidos e redefinidos de forma covariante terão todos a mesma assinatura.

Os métodos nos quais o programador quer usar covariância são marcados com o novo modificador **covariant**, quer se trate de uma definição ou redefinição. O uso deste novo modificador também implica que se proíbe sobrecarga (*overloading*) do nome do método envolvido. Todos os métodos que não estejam marcados com o modificador atrás referido serão tratados como métodos Java normais. A validação dos tipos dos argumentos nos métodos covariantes não será feita de forma estática, mas sim dinamicamente, tal como acontecia na versão original da linguagem Eiffel. No entanto, e como é sempre preferível a detecção de erros em tempo de compilação, existem diversas validações feitas estaticamente pelo tradutor e que poderão dar origem a mensagens de erro que desde logo inviabilizam a tradução para Java puro. Gera-se também, em certos casos, código Java auxiliar que será concatenado à tradução original, e que irá permitir que o compilador de Java possa validar certos aspectos relativos à covariância.

Para melhorar ainda mais a expressão da covariância na linguagem JavaCO, também se introduzirá um novo tipo genérico chamado **This**. Este tipo representará o tipo de **this**, palavra reservada que, usada em Java no corpo dum método, denota o objecto que recebeu a

mensagem. Este será um tipo polivalente, cujo significado será diferente consoante a classe ou subclasse em que seja interpretado. Quando um parâmetro dum método definido ou herdado tem tipo **This**, isso significa que esse parâmetro só admite objectos com o mesmo tipo dinâmico de **this**. A introdução deste novo mecanismo visa explorar ao máximo o potencial da covariância e vem "imitar" o mecanismo equivalente da linguagem Eiffel, que usa a sintaxe "*like Current*" [Meyer 06].

Na linguagem JavaCO, o tipo **This** também pode ser usado para tipificar variáveis, tanto locais como de instância, e resultados de métodos.

A obtenção de *output* em formato Java puro a partir de um programa em JavaCO é possível através do tradutor que foi implementado. Para isso foi usada a ferramenta SableCC [Gagnon 98].

O processo de tradução pode ser dividido em três fases principais: na primeira fase, é analisada a AST, recolhida informação e são efectuadas algumas validações; na segunda fase completa-se a informação sobre as classes com base nos dados recolhidos na etapa anterior e efectuam-se as restantes validações; na terceira fase, procede-se à alteração da AST, com vista à obtenção do código Java puro final.

Esta dissertação, inserida na área de Construção e Análise de Software, visa contribuir com a oferta de uma solução que permita aos programadores usufruir de uma das características mais marcantes da linguagem Eiffel, num ambiente de programação Java.

1.4. Estrutura da dissertação

A presente dissertação está organizada da seguinte forma:

O Capítulo 2 introduz a linguagem Eiffel, que esteve na base da motivação para a realização deste trabalho.

O Capítulo 3 descreve as principais noções relativas a Sistemas de Tipos, suas utilidades e principais características. Também introduz o tema da Covariância no âmbito deste trabalho.

No Capítulo 4 são descritas de forma sucinta as três ferramentas consideradas para a implementação do tradutor, e justifica-se a escolha do SableCC.

O Capítulo 5 apresenta de forma genérica o trabalho efectuado, desde os objectivos à implementação, passando pelas motivações, princípios orientadores e mecanismos introduzidos relativamente à linguagem Java. Este capítulo prepara o terreno para os capítulos seguintes, nos quais se entra em mais detalhes, tanto ao nível na discussão dos conceitos, como na apresentação das técnicas de implementação usadas.

O Capítulo 6, o mais vasto, é dedicado ao estudo de parâmetros Covariantes de tipos não genéricos. São abordadas as mais importantes técnicas e estratégias usadas na implementação e tradução, estruturas de dados usadas e o algoritmo para a escolha dos melhores tipos para substituição.

No Capítulo 7 é descrito o novo tipo genérico **This** para uso nos argumentos de métodos covariantes, suas propriedades e implicações na tradução.

O Capítulo 8 descreve o alargamento do uso do tipo **This** para tipificar variáveis e o retorno de métodos, nomeadamente a forma de validação, o método adoptado para essa validação e seu funcionamento.

No Capítulo 9 aborda-se a Covariância relativamente aos tipos genéricos e parâmetros de tipo, onde é dada a devida importância às limitações encontradas no que diz respeito à implementação deste mecanismo quando estamos perante estes tipos.

O Capítulo 10 descreve os principais mecanismos e características que poderiam ser adicionados ao tradutor, do ponto de vista de um Trabalho Futuro.

Por fim, no Capítulo 11 são apresentadas as Conclusões e a análise crítica ao trabalho desenvolvido ao longo da dissertação.

2. Eiffel

Sendo o Eiffel uma linguagem muito bem documentada, quase a totalidade da informação deste capítulo foi retirada da "bíblia" da linguagem, escrita pelo seu criador Bertrand Meyer [Meyer 06], bem como de uma introdução oficial ao Eiffel [Eiffel web 1], do tutorial oficial [Eiffel web 2] e da documentação online sobre os métodos e linguagem Eiffel [Eiffel web 3].

A Linguagem de Programação Eiffel apresenta-se como sendo uma linguagem orientada por objectos pura, desenhada com vista à reutilização, baseada em classes e que suporta herança múltipla e repetida. Além disso, possui um poderoso sistema de verificação de tipos, classes parametrizadas, ligação dinâmica (*dynamic binding*), recolha de lixo (*garbage collection*) e tratamento de excepções.

O Eiffel, talvez mais que outras linguagens de programação, remove complexidades de programação desnecessárias sem limitar a capacidade de exprimir a complexidade de software inerente. Tendo como focos principais a **correcção** e **reutilização** [Meyer 97], o Eiffel consegue exprimir abstracções de uma forma clara, utilizado uma sintaxe que se revela simples de aprender e utilizar. A linguagem pode ser caracterizada como sendo simples, eficiente, poderosa e portátil. Serão estas as razões pela qual é considerada uma linguagem de sucesso, sendo usada nas mais variadas áreas, como as finanças, saúde, telecomunicações, aeroespaço e jogos de vídeo.

2.1. Princípios de desenho

O objectivo do Eiffel consiste em ajudar na especificação, desenho, implementação e modificação de sofware de qualidade. Esta finalidade, obtenção de qualidade em software, é uma combinação de vários factores [Meyer 97]. Tendo em conta o actual estado da indústria, aqueles que apresentam maior necessidade de ser melhorados são a **reutilização** (*reusability*), **expansibilidade** (*extendibility*) e **fiabilidade** (*reliability*). Outros factores também importantes são a **eficiência** (*efficiency*), **interoperabilidade** (*openness*) e **portabilidade** (*portability*).

- **Reutilização** – capacidade de produzir componentes que podem ser usados em diferentes aplicações. A abordagem do Eiffel contempla a presença de bibliotecas largamente usadas que complementam a linguagem, bem como um suporte para a criação de novas bibliotecas.
- **Expansibilidade** – capacidade de produzir software fácil de modificar. O principal meio de apoio à expansibilidade no Eiffel é o desenho cuidadoso do mecanismo de herança. Ao contrário de outras linguagens, como o Smalltalk, que não suportam herança múltipla, a abordagem do Eiffel depende deste mecanismo para combinar várias abstracções numa só.
- **Fiabilidade** – capacidade de produzir software correcto e robusto, isto é, livre de erros. Cruciais para este esforço, são a presença de tipificação estática (*static typing*); asserções e todo o mecanismo de Desenho por Contracto, permitindo especificar precisamente o que o software deve fazer; tratamento de excepções disciplinado; recolha de lixo automática; abordagem clara de herança; uso por defeito de ligação dinâmica (*dynamic binding*), o que assegura que todas as chamadas usarão a versão correcta de cada operação; e a simplicidade do desenho da linguagem.
- **Eficiência** – capacidade de um sistema de software para colocar o mínimo de exigências quanto possível sobre os recursos de hardware, como o tempo de processador e o espaço ocupado nas memórias interna e externa. O Eiffel permite aos programadores criarem sistemas capazes de correr em condições de tempo e espaço

semelhantes às de programas escritos em linguagens de baixo nível, normalmente dirigidos a uma implementação eficiente.

- **Interoperabilidade** – o Eiffel permite criar software capaz de cooperar com programas escritos noutras linguagens.
- **Portabilidade** – facilidade de transferir produtos de software para diferentes ambientes de hardware e software. O Eiffel garante portabilidade através de uma definição de linguagem independente da plataforma, de modo que a mesma semântica possa ser suportada em diferentes plataformas.

2.2. Desenho orientado a objectos

A linguagem Eiffel usa o paradigma de programação orientada a objectos, que consiste no uso de objectos – estruturas de dados constituídas por campos de dados (variáveis) e métodos, capazes de interagir entre si – para a construção de aplicações e sistemas de software. Este tipo de programação consiste na construção de sistemas de software como um conjunto estruturado de implementações de tipos de dados abstractos, as chamadas classes.

Os seguintes pontos merecem destaque tendo em conta a definição de desenho orientado a objectos:

- A ênfase deste tipo de desenho está na estruturação de um sistema em torno dos tipos de objectos por ele manipulado conjuntamente com a reutilização de estruturas de dados e suas respectivas operações associadas;
- Os objectos são descritos como exemplos de tipos de dados abstractos, isto é, estruturas de dados conhecidas por uma interface oficial, e não através da sua representação;
- As classes, unidades modulares básicas, descrevem a implementação de um tipo abstracto de dados;
- As classes devem ser desenhadas como unidades interessantes e úteis por si próprias, isto é, independentemente do sistema onde estão integradas, bem como reutilizáveis

pelos mais diversos sistemas. Podemos então ver a construção de software como um agrupamento e interacção entre classes existentes.

2.3. Desenho por Contracto

O conceito de Desenho por Contracto é central ao Eiffel [Meyer 97]. Esta é uma abordagem ao desenho de software que prevê que os programadores devem definir especificações de interface formais e precisas para componentes de software. Estas especificações são introduzidas nas rotinas através de asserções, que expressam a exactidão das condições e podem ser de vários tipos:

- **pré-condições** – expressam requisitos que os clientes devem satisfazer cada vez que chamam a rotina. As pré-condições são introduzidas pela palavra-chave `require`;
- **pós-condições** – expressam condições que a rotina garante no retorno, caso as pré-condições tenham sido satisfeitas à entrada. As pós-condições são introduzidas pela palavra-chave `ensure`;
- **invariantes** – as invariantes da classe são restrições que devem ser satisfeitas por todas as instâncias da classe, quando estas são acessíveis a partir do exterior: após a criação, ou depois de uma chamada de uma rotina da classe. Introduzidas pela palavra-chave `invariant`, as invariantes representam restrições de consistência geral impostas em todas as rotinas da classe.

A ideia principal do Desenho por Contracto foca-se numa metáfora em como os elementos de um sistema de software colaboram uns com os outros, na base de obrigações e benefícios mútuos. A metáfora vem da vida empresarial, onde um fornecedor e um cliente estabelecem um contracto entre si, onde cada um tem obrigações a cumprir e benefícios a extrair.

Esta capacidade de verificar asserções fornece ao Eiffel um poderoso mecanismo de teste e *debug*, principalmente devido às mais importantes e utilizadas bibliotecas da linguagem, que estão protegidas por asserções cuidadosamente desenhadas.

As asserções são também uma ferramenta indispensável para a documentação de componentes de software reutilizável. Aqueles que são utilizados em larga escala, vêm munidos de uma documentação precisa que descreve o comportamento do mesmo: que restrições o componente espera ver satisfeitas (pré-condições), o que garante no retorno (pós-condições) e que estados/condições mantém invioláveis (invariantes).

2.4. A arquitectura do software Eiffel

Como já foi referido, as unidades modulares básicas do Eiffel são as *classes*. Para se manter o desenvolvimento de software organizado, as classes podem ser agrupadas em *clusters*, que não são uma construção sintáctica da linguagem mas sim uma convenção padrão organizacional. Os sistemas executáveis resultam da combinação de classes de um ou mais *clusters*.

A base da estrutura do software Eiffel pode então ser definida pelos conceitos:

- Classe, como unidade básica modular;
- *Cluster*, como um agrupamento lógico de classes;
- Sistema, como um ajuntamento de uma ou mais classes para produzir uma unidade executável.

Uma aplicação Eiffel é tipicamente organizada com cada classe num ficheiro separado, e cada *cluster* numa directoria contendo ficheiros de classes. Segundo esta organização, *subclusters* são subdirectorias. De acordo com convenções organizações standard, *pessoa.e* deverá ser o nome de um ficheiro que define a classe Pessoa.

As classes não são apenas a unidade modular da decomposição de software: servem também como base para os tipos em Eiffel.

Esta vista dupla é essencial à compreensão da noção de classe, bem como dos princípios de construção de software orientado a objectos:

- Como unidade de decomposição, uma classe é um módulo, isto é, um grupo de **serviços** relacionados, agrupados dentro de uma unidade nomeada.
- Como um tipo, uma classe é a descrição de elementos de dados semelhantes em tempo de execução, ou objectos, denominados de **instâncias** da classe.

As classes em Eiffel são compostas por *features*, que são semelhantes a "membros", "atributos" ou "métodos" noutras linguagens de programação orientadas por objectos. As classes também definem os seus invariantes (desenho por contracto), e podem conter outras propriedades, como por exemplo uma secção de "notas" para documentação e metadados.

Exemplo de uma classe:

```
class
        PESSOA
create
        make
feature  -- Inicialização
        make( s: STRING; x: CHARACTER) is
                -- criar uma pessoa com nome 's' e género 'x'
        do
                name := s
                genero = x
        end

feature  -- Acesso
        name : STRING

feature  {NONE} -- Implementação (Este atributo é privado)
        genero : CHARACTER
end;
```

O acesso às *features* utiliza a notação de ponto, da forma *alvo.nome_da_feature*, possivelmente seguida de uma lista de argumentos entre parênteses. Existem dois tipos de *features*:

- **Rotinas**, que representam computações aplicáveis a instâncias da classe.
- **Atributos**, que representam itens de dados associados a essas instâncias.

As Rotinas são divididas em procedimentos (comandos, que não retornam nenhum valor) e funções (*queries*, que retornam um valor).

As *features* podem estar disponíveis para todos os clientes das classes, em que a palavra-chave `feature` é introduzida sem qualquer qualificação. Se pretendermos que a *feature* não esteja disponível para os clientes da classe, esta será introduzida por `feature { NONE }`. O que aparece entre chavetas é uma lista de classes para as quais a *feature* em questão se encontra disponível; *NONE* é uma classe especial da Biblioteca Kernel, que não tem instâncias, tornando a *feature* secreta e apenas disponível localmente para outras rotinas da classe em questão.

Em Eiffel existe uma distinção entre tipos referência (cujos valores são acedidos usando apontadores) e tipos expandidos (cujos valores são guardados directamente nas variáveis).

Uma classe *X* pode tornar-se cliente da classe *PESSOA*, declarando uma ou mais entidades do tipo *PESSOA*:

```
p: PESSOA
```

Uma entidade declarada como sendo de um tipo referência, como é o caso de `p`, pode em qualquer altura durante a execução ficar ligada a um objecto. Uma entidade é dita *void* se não se encontra ligada a nenhum objecto, sendo este o tipo das entidades por defeito após a inicialização. Para obter objectos em tempo de execução, uma rotina `r` que apareça na classe cliente *X*, deve usar uma instrução de criação:

```
create p
```

Em Eiffel qualquer operação é relativa a um certo objecto. Um cliente invoca a operação específica para este objecto escrevendo a correspondente entidade à esquerda do ponto. Dentro da classe, a instância corrente sobre as quais normalmente se efectuam operações mantém-se implícita. No entanto, se for necessário denotar o objecto corrente explicitamente, podemos fazê-lo através da entidade especial Current.

Ao contrário de outras linguagens orientadas por objectos, em Eiffel não é permitida a afectação de campos de objectos, excepto dentro das rotinas de um objecto.

Princípio de Acesso-Uniforme: do ponto de vista de um cliente de software que faz uma chamada a uma *feature* de uma classe, não deve fazer diferença se a *query* é um atributo (campo de dados em cada objecto) ou uma função (algoritmo). Por exemplo, veiculo.velocidade pode ser um atributo, acedido a partir da representação do objecto; ou pode ser calculado por uma função que divide a distância pelo tempo. A notação é a mesma em ambos os casos, o que torna fácil alterar a representação sem afectar o resto do software.

Uma classe pode ser definida com **`deferred class`** em vez de apenas com **`class`** para indicar que a classe corresponde a um conceito abstracto e portanto não pode ser directamente instanciada. Estas classes são chamadas de classes abstractas noutras linguagens orientadas por objectos.

2.5. Entidades genéricas

Uma das técnicas para assegurar **reutilização** e **expansibilidade** na construção de componentes de software (classes) como implementações de tipos de dados abstractos são as **entidades genéricas** (*genericity*).

Criar uma classe genérica consiste em atribuir-lhe parâmetros genéricos, que representam tipos desconhecidos, como nos exemplos:

ARRAY [G]

LIST [G]

Estas classes representam estruturas de dados, contendo objectos de um certo tipo. O parâmetro genérico formal *G*, denota esse tipo.

Este mecanismo (entidades genéricas) apresenta duas variantes: não restringida (a dos exemplos anteriores) e restringida. Esta última permite que uma classe coloque exigências específicas sobre os parâmetros genéricos possíveis. De seguida é apresentado um exemplo, onde o parâmetro deve herdar de uma dada classe:

```
class HASH_TABLE [ G, KEY -> HASHABLE ]
```

Neste exemplo, uma derivação *HASH_TABLE [INTEGER, STRING]* só é válida se *STRING* herdar de *HASHABLE* (o que de facto acontece nas bibliotecas típicas de Eiffel). Dentro da classe, tendo *KEY* restringido por *HASHABLE* significa que para x: *KEY* é possível aplicar a x todas as funcionalidades de *HASHABLE*, como x.*hash_code*.

Não seria possível, sem entidades genéricas**,** termos verificação de tipos estática numa linguagem orientada por objectos realística.

2.6. Herança

A herança é um mecanismo de generalização fundamental que torna possível definir uma nova classe por combinação e especialização de classes existentes, em vez de partir do zero. Desta forma, a subclasse criada herda as *features* da ou das classes 'pai', evitando assim a criação repetida de atributos e funções semelhantes. A cláusula **inherit** lista todos os 'pais' da nova classe, dos quais é considerada herdeira.

A linguagem Eiffel suporta herança simples e múltipla. Na primeira apenas uma classe é herdada, enquanto na herança múltipla uma classe pode ter tantos 'pais' quanto for necessário. Segue-se um exemplo de herança simples:

```
class ALUNO inherit
        PESSOA
```

```
feature
        ...
```

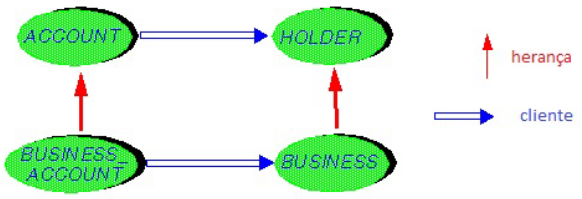
Por defeito, *ALUNO* vai incluir todas as *features* de *PESSOA*, às quais poderá adicionar as suas próprias caso seja necessário. Mas o mecanismo de herança é mais flexível, permitindo a *ALUNO* adaptar as funcionalidades herdadas de diversas formas, através das palavras-chave: **rename**, **export**, **undefine**, **redefine** e **select**.

Uma importante parte da herança que nos importa falar tendo em conta o tema deste trabalho, são as regras de adaptação dos tipos das *features* herdadas. Mais geral que tipo, é a noção de assinatura de uma *feature*, definida pelo seu número de argumentos, seus tipos e a indicação do tipo do resultado, caso a *feature* retorne um resultado.

Em muitos casos a assinatura de uma *feature* redeclarada mantém-se igual à original. Mas por vezes podemos querer adaptá-la à nova classe. Assumamos que a classe *ACCOUNT* tem as seguintes *features*:

```
owner: HOLDER

set_owner ( h: HOLDER ) is
        require
                not_void: h /= Void
        do
                owner := h
        end
```

Introduza-se um herdeiro *BUSINESS_ACCOUNT* de *ACCOUNT* que representa um tipo de contas bancárias, correspondente à classe *BUSINESS* herdeira de *HOLDER*:



Segundo a regra da **covariância**, na redeclaração de uma *feature*, tanto o tipo do resultado se a *feature* for uma *query*, como o tipo de qualquer argumento caso seja uma rotina, deve estar em conformidade com o tipo original declarado na classe 'pai'. Assim, todos os tipos variam na mesma direcção que a estrutura hierárquica.

Desta forma, devemos redefinir owner na classe *BUSINESS_ACCOUNT* para assegurar um resultado do tipo *BUSINESS*; a mesma redefinição da assinatura deve ser aplicado ao argumento de set_owner:

```
set_owner ( b: BUSINESS ) is
        -- novo corpo da rotina
end
```

Por vezes, o corpo da rotina irá ser exactamente igual, o que implicaria muita duplicação de código. Para evitar este problema, o Eiffel possui um mecanismo de **redeclaração ancorada**, que consiste em ter um tipo bem definido, que pode ser usado, neste caso, para definir o tipo de um argumento de uma *feature*.

Um tipo **like** âncora pode ser usado em qualquer contexto no qual âncora tenha um tipo bem definido. Assim, assumindo que *T* é o tipo de âncora, o tipo **like** âncora significa que na classe onde aparece, **like** âncora tem o mesmo significado que *T*. A diferença é que nos descendentes da classe, se uma redefinição de tipo altera o tipo de âncora, qualquer entidade declarada **like** âncora será considerada também como redefinida.

A declaração de set_owner em *ACCOUNT*, deveria então ser a seguinte:

```
set_owner ( h: like Current) is
require
   not_void: h /= Void
do
   owner := h
end
```

Desta forma, a classe *BUSINESS_ACCOUNT* apenas necessita de redefinir o tipo de owner (para *BUSINESS*). Não há assim necessidade de redefinir set_owner, a não ser que se queira alterar a implementação ou asserções.

2.7. Tipos e subtipos

O Eiffel é uma linguagem de programação orientada por objectos pura. Significa isto que até os tipos básicos como o *INTEGER* e o *BOOLEAN*, são objectos.

Esta linguagem é explicitamente tipificada para uma maior fiabilidade e legibilidade. Todas as entidades são declaradas como sendo de um certo tipo, que pode ser um tipo de referência ou um tipo expandido. A diferença é que, para um tipo de referência, o caso mais comum, os valores possíveis para uma entidade são referências para objectos; para um tipo expandido, os valores são objectos.

Um dos mais importantes grupos de tipos expandidos é o dos tipos básicos: *INTEGER*, *REAL*, *DOUBLE*, *CHARACTER* e *BOOLEAN*. Claramente, o valor de uma entidade declarada com o tipo *INTEGER* deve conter um inteiro e não uma referência para um objecto que contém um valor inteiro.

O sistema de tipos do Eiffel é uniforme e consistente visto que todos os tipos, incluindo os tipos básicos, são definidos em classes.

O Eiffel permite tipos de retorno e de parâmetros covariantes na redefinição de métodos (a covariância é validada estaticamente). Isto é possível porque a linguagem usa uma noção de subtipo pouco ortodoxa e que só por si não garante segurança em tempo de execução. Concretamente, quando um objecto de uma subclasse é tratado presumindo-se que pertence a uma classe mais geral (polimorfismo), há teoricamente a possibilidade de erros de execução.

Actualmente, o Eiffel usa um compilador muito sofisticado que detecta em tempo de compilação todas as situações em que a covariância cria o problema atrás descrito. Nessas situações, o programador é chamado a corrigir o código. No entanto, no Eiffel “clássico”, de

há alguns anos atrás, os erros (chamados *catcalls*) [Meyer 05] provocados pela covariância eram mesmo apanhados em tempo de execução e davam origem a excepções.

2.8. Conformidade

Esta é considerada a mais importante característica do sistema de tipos Eiffel, no sentido em que determina quando um tipo pode ser usado no lugar de outro.

Esta propriedade tem um papel tão preponderante visto que é o seu uso que torna segura tanto a afectação de variáveis como a passagem de argumentos: para `x` do tipo *T* e `y` do tipo *V*, a instrução `x := y`, e a chamada `rotina (y)` com `x` como argumento formal, só serão válidas se *V* for compatível com *T*, significando que está em *conformidade* ou que *converte* para *T*.

A *conformidade*, assim como todo o sistema de tipos, tem como base a herança. Assim sendo, as condições básicas para *V* estar em *conformidade* com *T* são as seguintes:

- A classe base de *V* deve ser deve ser um descendente da classe base de *T*.
- Se *V* é um tipo derivado genericamente, os seus parâmetros genéricos deverão estar em *conformidade* com os correspondentes em *T*. Quer isto dizer que *B [Y]* está em *conformidade* com *A [X]* apenas se *B* está em *conformidade* com *A* e *Y* com *X* (note que isto significa que as classes genéricas do Eiffel são covariantes nos seus argumentos).
- Se *T* é um tipo expandido, a herança não está envolvida. Desta forma, *V* só poderá ser do próprio tipo *T*.

Em outras linguagens de programação, afirmar que o tipo *A* está em *conformidade* com o tipo *B*, é o mesmo que dizer que *A* é subtipo de *B*. Sendo esta uma designação mais comum, esta propriedade será designada de ***subtipo*** ao longo deste trabalho.

2.9. Outros mecanismos e funcionalidades

2.9.1. Erradicar chamadas void

Sendo uma chamada a uma *feature* a operação básica de computação na programação orientada por objectos, está sempre presente a possibilidade da *feature* ser aplicada a uma variável que não representa um objecto, mas sim uma referência, que por sua vez pode ser *void*, não contendo nenhum objecto anexado.

Estas chamadas são uma fonte de instabilidade e de bloqueios em programas orientados por objectos.

O Eiffel fornece um conjunto de técnicas que garantem a ausência deste tipo de chamadas em tempo de execução. Estas técnicas consistem em distinguir as referências que podem tomar valor *void* das que não permitem que tal aconteça. Assim, considera-se que uma referência com um nome usual, `T`, que não pode ter valor *void*, é *attached*, sendo "Attached" uma propriedade estática. Para se permitir valores *void*, usa-se `?T`, que representa uma referência *detachable*.

Esta regra assegura que uma entidade `x: T`, declarada do tipo *attached*, nunca poderá conter valor *void* em tempo de execução. Em particular, não se pode afectar `x` com um valor *detachable*.

2.9.2. Overloading

Em Eiffel não é permitido *overloading*. Apenas existe uma *feature* com um determinado nome dentro de uma classe. Ao evitar uma causa de ambiguidade sobre que *feature* é invocada por uma chamada, esta medida melhora a legibilidade das classes.

2.9.3. Agentes

O mecanismo de agentes Eiffel serve para encapsular operações em objectos. Desta forma, outras partes do software podem mais tarde percorrer esses objectos para encontrar as operações, e executá-las.

Este mecanismo pode ser útil em diversas áreas, como por exemplo: na programação de interfaces gráficas, em que podemos associar um agente a um determinado evento na interface, que poderá desencadear a operação associada ao agente; em iterações em estruturas de dados, se quisermos aplicar uma operação a todos os elementos de uma estrutura como uma lista, pode passar-se ao mecanismo de iteração um agente que represente essa operação.

2.9.4. Excepções

O tratamento de excepções em Eiffel baseia-se nos princípios de desenho por contracto, já falados neste capítulo. Por exemplo, uma excepção é lançada quando uma chamada de uma *feature* não satisfaz uma pré-condição, ou quando uma *feature* não consegue assegurar a validade de uma pós-condição.

Existe a possibilidade de uma *feature* tratar a excepção, sendo a operação que a trata definida pela palavra-chave `rescue`. Por sua vez, a palavra-chave `retry` executa a rotina de novo.

2.10. Eiffel vs Java

Como já foi referido, Eiffel é uma linguagem bastante apreciada no universo dos programadores, devido às suas características que se enquadram nas teorias sobre orientação a objectos de Bertrand Meyer. Sendo uma linguagem que permite uma expressividade particularmente interessante, foi adoptada como base para a realização deste trabalho. O objectivo principal deste trabalho parte então da motivação de usufruir das mais importantes características do Eiffel num ambiente de programação em Java.

Tendo em conta a abordagem sistemática à linguagem Eiffel que é feita ao longo deste documento, com vista a uma implementação da linguagem JavaCO à imagem do Eiffel, vamos aqui fazer alusão às principais características da linguagem Eiffel, e consequente comparação em relação às propriedades correspondentes que existam na linguagem Java.

Eiffel	Java
Covariância dos argumentos em redefinição de métodos Esta linguagem possui um poderoso mecanismo de herança. Uma das suas propriedades passa por permitir que se especializem argumentos dos métodos na sua redefinição em subclasses.	A linguagem Java não permite tipos mais específicos nos argumentos de métodos redefinidos, em comparação aos tipos declarados nos argumentos dos métodos na superclasse.
Abordagem completamente orientada a objectos As classes são usadas em Eiffel como base única para a Estrutura dos Módulos e Sistema de Tipos. Apresenta um sistema de tipos consistente, dado que todos os tipos são baseados em classes, incluindo os tipos básicos como inteiros, booleanos, caracteres, etc.	**Linguagem não puramente orientada a objectos** Em Java nem tudo são classes/objectos. Esta linguagem possui um grupo especial de oito tipos de dados, normalmente chamados de tipos primitivos, que foram introduzidos na linguagem de forma a melhorar o desempenho. São eles: *boolean*, *char*, *byte*, *short*, *int*, *long*, *float* e *double*. Estes tipos não podem ser estendidos, nem permitem a adição de novos métodos.
Herança múltipla Possibilidade de definição de classes como extensão ou especialização de outras classes. Em Eiffel uma classe pode herdar de quantas classes for necessário. É permitida a renomeação de *features* nas classes herdadas.	**Herança simples** A linguagem possui o mecanismo de herança, sendo permitida a extensão e especialização das classes. No entanto, uma classe só pode herdar de uma única classe.
Desenho por Contracto e Asserções Pré-condições, pós-condições e invariantes; Constituem um poderoso mecanismo, que tem como objectivo principal a correcção dos programas. É também factor precioso para a documentação.	**Asserções** A linguagem Java possui um mecanismo de asserções que permite assegurar a veracidade de uma expressão booleana em tempo de execução. As asserções não são herdadas ao contrário dos contractos do Eiffel.
"like Current" Declaração que denota um tipo baseando-se na classe corrente. Esta declaração é particularmente interessante e útil da redeclaração de métodos em classes herdadas.	A linguagem Java não possui nenhum mecanismo semelhante.

Tipos genéricos Uma das técnicas fundamentais para assegurar a criação de software reutilizável e expansível. Permite a implementação de tipos de dados abstractos, criando estruturas flexíveis, contendo objectos de um certo tipo, sendo que o parâmetro genérico denota esse tipo. As entidades genéricas em Eiffel podem ser restringida ou não restringida. A variante restringida deste mecanismo permite colocar requisitos específicos em possíveis parâmetros genéricos.	**Tipos genéricos** Técnica igualmente disponível na linguagem Java. Também aqui podemos aplicar restrições, introduzindo fronteiras aos tipos que cada genérico aceitará.
Conformidade de tipos Característica que determina quando um tipo pode ser usado no lugar de outro. Fundamental na segurança do sistema de tipos Eiffel, no sentido em que evita erros de tipos tanto na afectação de variáveis como na passagem de argumentos, assegurando a compatibilidade entre os tipos a tratar.	**Subtipos** Relação entre classes, diz-nos quando podemos passar um objecto de uma dada classe a um método de outra classe, ou afectar uma variável de uma classe com um objecto de outra classe.
Erradicação de chamadas *void* A introdução da noção de tipos "Attached" e "Detachable", sendo que só os últimos permitem *void* como valor possível, agregada a outras técnicas de programação, como o recurso a CAP (Certified Attachment Patterns), que são esquemas de programas que são demonstráveis como *void-safe*, permite a erradicação de desreferenciações nulas.	Mecanismo não existente em Java.
Tipificação estática e dinâmica A verificação de tipos é, na sua maioria, efectuada em tempo de compilação (estaticamente). No entanto, existem verificações que são feitas dinamicamente.	**Tipificação estática e dinâmica** Além da verificação estática de tipos, a linguagem Java também efectua algumas verificações dinamicamente, nomeadamente quando se recorre ao operador `instanceof`, que verifica em tempo de execução se um objecto é uma instância de uma determinada classe, e quando se recorre a *casts*.
Tratamento de Excepções As excepções em Eiffel podem surgir caso um contracto seja violado ou devido a condições anormais reportadas pelo hardware ou sistema operativo. Esta linguagem permite a detecção e tratamento de excepções dentro das rotinas.	**Tratamento de Excepções** A linguagem Java também permite o tratamento de excepções dentro dos métodos.

Ligação Dinâmica (*Dynamic Binding*) Aparece quando uma função membro de uma classe derivada é invocada com base no tipo dinâmico do objecto. O poder da programação orientada por objectos reside em grande parte neste mecanismo.	**Ligação Dinâmica (*Dynamic Binding*)** Esta propriedade existe em Java e é idêntica à da linguagem Eiffel.
Polimorfismo Propriedade que permite a definição de entidades flexíveis, que podem ficar anexadas a objectos de vários tipos em tempo de execução.	**Polimorfismo** O Java possui o mesmo recurso através da herança, permitindo que referências de classes mais abstractas representem o comportamento das classes concretas que referenciam, possibilitando o tratamento de vários tipos de maneira homogénea.
Agentes Mecanismo que aplica conceitos orientados por objectos à modelação de operações. Passa por encapsular operações em objectos, à semelhança de *closures*.	A linguagem Java não tem directamente o conceito de Agente, mas também permite criar *closures* através de classes anónimas.
Garbage collection Mecanismo que gere automaticamente a recuperação de memória ocupada por objectos que já não estão em uso pelo programa.	**Garbage collection** A linguagem Java também possui um mecanismo que de forma automática gere a libertação de memória ocupada por objectos que já não se encontram em uso.
Classes "Deferred" Uma classe é "deferred" se contém pelo menos uma *feature* "deferred". Significa isto que não existe implementação dessa *feature* dentro desta classe, implementação essa que aparecerá em eventuais descendentes da mesma. É o nome dado em Eiffel a classes abstractas.	**Classes abstractas** Mecanismo igual ao existente em Eiffel.

3. Sistemas de Tipos

Relativamente ao desenvolvimento de software, os dados usados são tão importantes como os programas em si. No âmbito da programação, dados são aquilo que pode ser armazenado, avaliado directamente, passado como argumento ou retornado como resultado de uma função, ou seja, são todas as entidades que existem durante uma computação.

Um Sistema de Tipos divide os valores de um programa em conjuntos chamados tipos, tendo a capacidade e objectivo de tornar ilegais certos comportamentos do programa, baseando-se nos tipos atribuídos no processo de tipificação. Se um Sistema de Tipos classificar o valor "olá" como uma cadeia de caracteres e o valor 2 como um número inteiro, torna assim ilegal um programa que some, por exemplo, ambos os valores.

Um tipo pode ser primitivo, no sentido em que contém pouca informação, como um inteiro, um valor booleano (verdadeiro ou falso), ou uma palavra (*String*); pode também ser composto, contendo maior quantidade de informação, como por exemplo um conjunto de tipos primitivos, e até funcionalidades.

Cada linguagem possui um mecanismo para verificar a correcção de um Sistema de Tipos [Pierce 02]. Este processo de verificação das restrições de tipos pode ser efectuado durante a compilação de um programa (**verificação de tipos estática** ou **tipificação estática**) ou durante a execução do programa (**verificação de tipos dinâmica** ou **tipificação dinâmica**). Estas duas variantes de análise de tipos de um programa serão abordadas mais à frente.

Podemos ver os tipos como uma forma de representar abstracções, sendo esta a capacidade mais poderosa das linguagens orientadas por objectos. Os tipos associam objectos a classes da linguagem de programação. Consideremos o exemplo de uma classe Pessoa, que

representa isso mesmo, uma pessoa genérica. Esta classe possuirá, entre outros, atributos para a data de nascimento, para o nome e para a localidade de nascimento, bem como funções que devolvem a idade (calculada através da data corrente e da data de nascimento) e o nome. Se quisermos entrar em maior nível de detalhe sobre uma pessoa, por exemplo um estudante, podemos criar uma classe Estudante, que herda do tipo Pessoa, adquirindo todos os seus atributos e métodos. Ao novo tipo poderemos adicionar novos atributos, como por exemplo a universidade e o número de aluno. Isto permite a criação de relações de Subtipos, algo considerado por muitos como uma das mais importantes características de uma linguagem de programação.

O objectivo fundamental de um Sistema de Tipos consiste em prevenir a ocorrência de erros de execução durante a execução de um programa, como defende Cardelli [Cardelli 02].

Pode considerar-se que um sistema de tipos calcula uma aproximação estática dos comportamentos que os termos de um programa terão em tempo de execução. Geralmente, os tipos atribuídos a esses termos são calculados de forma composta, sendo que o tipo de uma expressão depende apenas dos tipos das suas subexpressões.

3.1. Para que servem os Sistemas de Tipos

Segundo Benjamin C. Pierce [Pierce 02], os Sistemas de Tipos são importantes no que diz respeito a:

- **Detectar Erros**

O mais óbvio benefício da verificação de tipos estática é a possibilidade de detectar com antecedência alguns erros de programação. Erros que são detectados na altura devida podem ser tratados de imediato, evitando-se desta forma ter que, mais tarde, percorrer todo o código à procura deles. Além disso, os erros podem muitas vezes ser localizados com mais precisão

durante a verificação do que em tempo de execução, altura em que os seus efeitos podem não saltar logo à vista do programador até que algo comece a correr mal.

Tirar o máximo partido de um sistema de tipos requer geralmente alguma atenção por parte do programador, bem como uma vontade de usar de forma apropriada as facilidades fornecidas pela linguagem. Por exemplo, um programa complexo que faça uso de listas para codificar todas as suas estruturas de dados, não irá tirar tanto partido do compilador como um que defina um tipo de dados ou um tipo abstracto diferente para cada uma. Os sistemas de tipos expressivos oferecem ao programador inúmeros meios para codificação de informação sobre estruturas em termos de tipos.

Um verificador de tipos pode também ser uma ferramenta de manutenção indispensável. Por exemplo, se houver a necessidade de se alterar a definição de uma estrutura de dados complexa, não será necessário procurar, em todo o programa, o código que envolve essa estrutura a fim de proceder à sua reparação. Uma vez alterada a declaração do tipo de dados, todas essas partes de código se tornam inconsistentes em relação aos tipos, podendo ser enumeradas apenas correndo o compilador e examinando onde a verificação de tipos falha.

- **Abstracção**

Os sistemas de tipos também suportam o processo de programação no sentido em que forçam uma programação disciplinada. No contexto da composição de software de larga escala, os sistemas de tipos são a mais importante ferramenta das linguagens de módulos, sendo usados para agregar e manter juntos os componentes destes complexos sistemas. Os tipos aparecem nas interfaces dos módulos, podendo estas ser vistas como o “tipo do módulo”, fornecendo as diversas funcionalidades que o mesmo oferece.

Estruturar sistemas complexos em termos de módulos com interfaces claras leva-nos a um estilo de desenho mais abstracto, onde as interfaces são desenhadas e discutidas independente das suas eventuais implementações. Um pensamento mais abstracto relativamente às interfaces geralmente conduz a um desenho melhor.

- **Documentação**

Os tipos são também bastante úteis na *leitura* dos programas. As declarações dos tipos em cabeçalhos de procedimentos e interface de módulos constituem uma forma de documentação, dando pistas úteis sobre o seu comportamento. Além disso, ao contrário de descrições incorporadas nos comentários, esta forma de documentação não pode ficar desactualizada, visto que é verificada durante todas as execuções do compilador. Este papel dos tipos é particularmente importante nas assinaturas dos módulos.

- **Segurança da linguagem**

Podemos definir linguagem segura como uma linguagem que protege as suas próprias abstracções. Todas as linguagens de alto nível fornecem abstracções de serviços máquina. Esta segurança de que se fala nas linguagens, refere-se à capacidade que a linguagem tem de garantir a integridade destas abstracções assim como de abstracções de alto nível introduzidas pelo programador usando as facilidades de definição da linguagem.

Cardelli [Cardelli 97], faz a distinção entre duas espécies de erros de execução: aqueles que fazem a computação terminar de imediato (***trapped errors***), como por exemplo a divisão por zero e o acesso a endereços ilegais, e aqueles que passam despercebidos (durante algum tempo), causando mais tarde um comportamento arbitrário (***untrapped errors***), como por exemplo aceder a um endereço legal de forma imprópria (por exemplo aceder a dados numa posição após o final de um *array* na ausência de verificação dos limites do mesmo em tempo de execução). Deste ponto de vista, considera-se que um fragmento de um programa é seguro se não causa a ocorrência de ***untrapped errors***.

3.2. Sistemas de tipos estáticos e dinâmicos

3.2.1. Tipificação Estática

Diz-se que uma linguagem usa tipificação estática quando a verificação de tipos é realizada em tempo de compilação. Nesta tipificação, os tipos encontram-se associados a variáveis, e não aos respectivos valores. Nas linguagens que usam esta tipificação, declaram-se os tipos das variáveis, dos argumentos e do resultado das funções, oferecendo desta forma uma base para o compilador efectuar as validações necessárias. Nestas linguagens os tipos são associados às expressões que aparecem no texto dos programas, sendo o próprio texto que é validado. Desta forma, aquando da execução, os tipos dos valores serão ignorados de uma forma geral, visto que a ausência de erros de tipo foi já garantida pelo compilador.

A verificação de tipos estática permite detectar muitos erros de tipos numa fase inicial do ciclo de desenvolvimento dos programas. Estes verificadores apenas avaliam a informação de tipos que pode ser determinada em tempo de compilação, sendo no entanto capazes de verificar que as condições examinadas se mantêm para todas as execuções possíveis do programa. A grande vantagem é que, desta forma, se elimina a necessidade de repetir estas verificações de tipos de cada vez que o programa é executado.

Sendo estáticos, estes sistemas de tipos são também, necessariamente, *conservadores*: os tipos podem representar informação incerta (podem representar, por exemplo, classes abstractas), correspondendo assim a "aproximações" dos valores que serão realmente usados na execução do programa, e que são imprevisíveis. Como forma de garantir a segurança, o compilador é forçado a assumir sempre o pior caso possível, mesmo que este acabe por não ocorrer. Imagine-se que se declara uma variável `t` de tipo Transporte, a qual recebe um avião. Se mais adiante tentarmos fazer levantar voo o transporte apontado por `t`, o compilador terá de considerar que nessa altura a variável poderá já não apontar para um avião, gerando um erro de compilação. No entanto, em tempo de execução, este código poderia não revelar qualquer tipo de problema.

Os especialistas em sistemas de tipos aprovam o *conservadorismo* da tipificação estática, argumentando que rejeitar alguns programas legítimos é preferível a aceitar um número aparentemente ilimitado de programas ilegítimos.

Dito de outra forma, estes sistemas de tipos *conservadores* conseguem provar categoricamente a ausência de alguns maus comportamentos nos programas, mas não conseguem provar a sua presença e, portanto, devem por vezes rejeitar programas que na realidade se comportam bem em tempo de execução. Segundo Benjamin C. Pierce [Pierce 02], por exemplo, um programa como

> **`if`** <teste complexo> **`then`** 10 **`else`** <erro de tipos>

será rejeitado como inválido, mesmo que aconteça que o <teste complexo> seja sempre avaliado como **true**, visto que uma análise estática não consegue determinar que isto vai acontecer. O comportamento *conservador* dos verificadores de tipos estáticos é vantajoso quando o <teste complexo> é raramente avaliado como **false**: um verificador de tipos estático pode detectar erros de tipos em caminhos de código que raramente são usados. Sem verificação estática de tipos, até mesmo testes que cubram 100% do código podem ser incapazes de descobrir tais erros. Estes testes de cobertura de código podem não detectar esses erros pois tem que se ter em conta a combinação de todos os sítios onde os valores são criados e todos os sítios onde um certo valor é usado.

Tal como acontece com uma vasta diversidade de termos partilhados por grandes comunidades, é também difícil apresentar uma definição precisa para “Sistema de Tipos”. Benjamin C. Pierce, “defensor” dos sistemas de tipos estáticos, apresenta a seguinte definição para o termo:

Um sistema de tipos é um método sintáctico tratável para provar a ausência de certos comportamentos do programa, através da classificação de frases de acordo com as espécies de valores que elas computam. [Pierce 02]

As mais utilizadas linguagens de programação com tipificação estática, não são formalmente ***type safe***. Normalmente contêm fraquezas ou excepções na especificação da própria linguagem que permitem que o programador escreva código que contorna as verificações

estáticas deste sistema de tipos, dando origem a uma ampla gama de problemas. Algumas destas linguagens disponibilizam técnicas que contornam ou corrompem o sistema de tipos. Estas operações podem trazer insegurança em tempo de execução, visto que podem causar comportamentos indesejados nos programas, devido a tipificações ou valores incorrectos aquando da execução dos mesmos.

3.2.2. Tipificação Dinâmica

Diz-se que uma linguagem usa tipificação dinâmica quando a maioria da sua verificação de tipos é realizada em tempo de execução. Ao contrário da tipificação estática, aqui os tipos são associados a valores, e não às variáveis. Nas linguagens que usam tipificação dinâmica não se declaram os tipos das variáveis, dos argumentos e dos resultados das funções. Assim, uma mesma variável pode conter valores de diferentes tipos, em diferentes momentos da execução dos programas. Estas linguagens associam os tipos aos valores que são usados em tempo de execução, sendo portanto necessária a existência de informação de tipo, nessa altura, associada a esses valores. Desta forma, o sistema de execução testa os tipos dos argumentos quando uma operação é aplicada aos mesmos.

Comparada com a tipificação estática, a tipificação dinâmica pode ser bastante mais flexível (permitindo, por exemplo, que os programas gerem tipos e funcionalidades baseados em informação recolhida em tempo de execução), embora à custa de menos garantias à priori. Isto deve-se ao facto de as linguagens tipificadas dinamicamente aceitarem e tentarem executar programas que seriam desde logo invalidados por um verificador de tipos estático.

A tipificação dinâmica pode resultar em erros de tipos em tempo de execução. Significa isto que, em tempo de execução, um valor pode ter um tipo inesperado, e uma operação sem sentido para esse tipo ser-lhe aplicada. Um *bug* deste tipo pode ser difícil de encontrar, visto que essa operação pode ocorrer bastante depois do local do programa que está a provocar o erro.

Por um lado, as verificações em tempo de execução podem ser potencialmente mais sofisticadas, visto que podem usar informação dinâmica para além de qualquer informação presente durante a compilação. Por outro lado, este tipo de verificações apenas consegue assegurar que as condições se mantêm numa execução particular do programa, sendo estas verificações repetidas em cada execução do programa.

Ao contrário do que acontece na tipificação estática, nos sistemas de tipo que utilizam a tipificação dinâmica não precisam de ser conservadores. Isto deve-se ao facto da sua influência ser exercida quando os programas são executados, altura na qual os argumentos exactos a ser utilizador pelas operações são conhecidos e encontram-se disponíveis para teste.

3.2.3. Tipificação Estática vs Tipificação Dinâmica

Sendo cada uma destas variantes de sistemas de tipos munida de características específicas, optar por uma delas requer a abdicação de algumas dessas propriedades, em detrimento de outras consideradas mais importantes.

As linguagens tipificadas estaticamente têm habitualmente de reduzir a expressividade da linguagem “base”, de modo a conseguirem prever alguns dos comportamentos dos programas em tempo de execução. Quando o verificador de tipos não consegue perceber se o uso dos tipos é correcto ou não, estas linguagens são forçadas a rejeitar programas que não sabem se são correctos ou não. Assim, o programador tem de rejeitar por vezes programas que não iriam dar erros, o que lhe diminui a liberdade de expressão de problemas correctos.

Por outro lado, os defensores da tipificação dinâmica acreditam que conseguem mais facilmente passar as suas ideias para os programas na ausência dos constrangimentos artificiais impostos pela tipificação estática, defendendo que os seus programas têm provado ser robustos e conter poucos erros.

Aparentemente é possível a obtenção de bons resultados seguindo qualquer uma destas escolas de programação, desde que se usem boas técnicas de desenvolvimento de software.

Uma das técnicas que se revela mais importante em qualquer das escolas de programação, é a escrita de muitos testes unitários, que servem para validar de uma forma sistemática todos os aspectos do software em desenvolvimento.

3.3. Relação de Subtipo

Em linguagens de programação, relação de subtipo é uma forma de polimorfismo de tipos no qual um **subtipo** é um tipo de dados que está relacionado com outro tipo de dados (o **supertipo**), pela noção de *substituibilidade*. Significa isto que as construções do programa, tipicamente subrotinas ou funções, escritas para operar sobre elementos do supertipo, podem também operar sobre elementos do subtipo.

Se A for subtipo de B, a relação de subtipo é normalmente escrita da forma A <: B, significando que qualquer termo do tipo A pode ser usado com segurança num contexto onde se espera um termo do tipo B.

3.3.1. Para que servem os Subtipos

Os subtipos permitem organizar os conceitos usados nos programas em diferentes níveis de abstracção, permitindo tratar objectos de um tipo mais específico como se fossem objectos de um tipo mais geral.

A abstracção é algo fundamental na forma como o ser humano raciocina. Abstrair significa ignorar deliberadamente aspectos particulares duma entidade ou dum conceito, com o objectivo de enfatizar os restantes aspectos, considerados de maior relevância. Quanto mais abstracto é um conceito, menor é a informação que lhe está associada e maior é o número de elementos que representa. Pelo contrário, quanto mais concreto é um conceito, maior é a informação que lhe está associada, sendo menor o número de elementos que representa.

Uma linguagem com Subtipos é, portanto, uma linguagem que se adapta à forma de pensar do programador, pois pensamos e falamos de tudo o que nos rodeia usando abstracções.

3.3.2. Propriedades da Relação de Subtipo

De seguida elucidam-se as regras que garantem a segurança dos Sistemas de Tipos, segundo Cardelli [Cardelli 97].

- **Reflexiva**

  Qualquer tipo é Subtipo dele próprio.

$$\frac{}{\boldsymbol{A} <: \boldsymbol{A}}$$

- **Transitiva**

  Se A é Subtipo de B e B é Subtipo de C, então A é Subtipo de C.

$$\frac{\boldsymbol{A} <: \boldsymbol{B} \quad \boldsymbol{B} <: \boldsymbol{C}}{\boldsymbol{A} <: \boldsymbol{C}}$$

- **Anti-Simétrica**

  Se A é Subtipo de B e B é Subtipo de A, então o tipo A é igual ao tipo B.

$$\frac{\boldsymbol{A} <: \boldsymbol{B} \quad \boldsymbol{B} <: \boldsymbol{A}}{\boldsymbol{A} = \boldsymbol{B}}$$

- **Relação de herança**

  Sendo A Subtipo de B, A herda todos os atributos e métodos de B. Assim, os registos de A são Subtipos dos registos de B, podendo A ter novos atributos e métodos não definidos na superclasse.

$$\frac{\boldsymbol{A_1} <: \boldsymbol{B_1} \ \ldots \ \boldsymbol{A_n} <: \boldsymbol{B_n}}{\{\boldsymbol{x_1}: \boldsymbol{A_1} \ \ldots \ \boldsymbol{x_n}: \boldsymbol{A_n} \ \ldots \ \boldsymbol{x_{n+m}}: \boldsymbol{A_{n+m}}\} <: \{\boldsymbol{x_1}: \boldsymbol{B_1} \ \ldots \ \boldsymbol{x_n}: \boldsymbol{B_n}\}}$$

- **Contravariância**

  *A intuição é que, se temos uma função f do tipo A → B, então sabemos que f aceita elementos do tipo A e retorna elementos do tipo B. Então, f também aceitará elementos de um **subtipo** A' de A, e os elementos que retorna podem ser considerados elementos de qualquer **supertipo** B' de B. Isto é, qualquer função de tipo A → B pode ser vista como sendo de tipo A' → B'.*

$$\frac{A' <: A \qquad B <: B'}{A \rightarrow B \; <: \; A' \rightarrow B'}$$

3.4. Covariância

Num Sistema de Tipos de uma linguagem de programação, uma regra de tipificação ou um operador de conversão de tipos é **covariante** se preserva a ordenação, ≤, de tipos, que ordena os tipos do mais específico para o mais geral, ou **contravariante** se inverte essa ordem, sendo a ordenação do mais geral para o mais específico. [F. Miller et al. 10]

Normalmente em linguagens orientadas por objectos, se uma classe B é subtipo de uma classe A, então cada função de B deve retornar um tipo igual ou mais restrito que o tipo retornado pela função correspondente na superclasse A; diz-se então que o tipo de retorno é covariante. Por outro lado, as funções de B devem aceitar um conjunto de argumentos igual ou mais amplo comparado com as respectivas funções de A; diz-se que o tipo dos argumentos é contravariante. O problema para as instâncias de B é como ser perfeitamente *substituível* para instâncias de A. A única forma de garantir segurança de tipos e *substituibilidade* é aceitar *inputs* iguais ou mais liberais que A, e ser igual ou mais restrito que A nos *outputs*.

Esta situação cria problemas em algumas situações, onde os tipos dos argumentos deveriam ser covariantes de forma a ajustarem-se aos requisitos da vida real. Suponhamos que temos

uma classe que representa uma pessoa. Uma pessoa pode ser vista por um doutor, pelo que esta classe deverá ter um método `consulta(Doutor d)`. Suponhamos agora que temos uma classe `Criança`, subclasse de `Pessoa`. Analogamente, temos uma classe `Pediatra`, sublcasse de `Doutor`. Sendo que as crianças apenas são consultadas por pediatras, seria útil reforçar este facto no sistema de tipos, reforçando consequentemente a expressividade do programa. No entanto, uma implementação simples falha: como uma `Criança` é uma `Pessoa`, `Criança::consulta(d)` poderá receber qualquer `Doutor`, e não apenas um `Pediatra` como pretendido.

Podíamos tentar mover o método `consulta()` para a classe `Doutor`, mas iríamos deparar-nos com o mesmo problema: Se um `Doutor` pode atender uma `Pessoa` e uma `Criança` é uma `Pessoa`, então continua a não haver maneira de reforçar o facto de que uma `Criança` deva ser vista por um `Pediatra` e que uma `Pessoa` que não é uma `Criança` não pode ser vista por um `Pediatra`, devendo ser atendida por outro `Doutor`.

A linguagem Eiffel, abordada no Segundo Capítulo deste documento, permite parâmetros covariantes em métodos redefinidos. O que torna isto possível é o facto da linguagem Eiffel usar uma noção de subtipo pouco ortodoxa, que pode ter algumas vantagens práticas mas não garante a segurança de tipos em tempo de execução. As “falhas” do sistema de tipos estático são compensadas por validações efectuadas em tempo de execução, que podem dar origem a excepções.

4. Ferramentas

Este capítulo visa uma sintética introdução a três ferramentas passíveis de serem usadas para a implementação do trabalho, bem como uma análise comparativa, em termos das necessidades e funcionalidades necessárias para essa mesma implementação.

4.1. JavaCC

Este subcapítulo foi escrito com base nas documentação oficial do JavaCC [Copeland 07], [JavaCC web 1] e [JavaCC web 2].

JavaCC (Java Compiler Compiler) é uma ferramenta geradora de analisadores sintácticos (*parsers)* e geradora de analisadores lexicais (*lexers*). Este programa lê a descrição de uma linguagem e gera código, escrito em Java, que irá ler e reconhecer a estrutura sintáctica de programas escritos nessa linguagem. Apresenta-se como uma ferramenta particularmente útil quando temos de escrever código para lidar com uma linguagem cuja estrutura é complexa, caso em que se pode tornar bastante difícil a criação manual de um *parser*.

O seu funcionamento pode ser descrito de uma forma simples: O *token manager* converte uma sequência de caracteres em objectos *Token*, sequência que será depois analisada pelo *Parser*. Os métodos usados pelo *token manager* para a conversão dependem obviamente da linguagem, sendo fornecidos pelo programador na forma de expressões regulares. Por sua vez, o *Parser* consome uma sequência de *tokens*, analisa a sua estrutura e produz a Árvore de Sintaxe Abstracta (cuja integridade, no JavaCC, está nas mãos do programador).

O JavaCC gera *parsers* top-down (descendentes recursivos). Este tipo de *parsers* apresenta algumas vantagens, como o uso de gramáticas bastante gerais, maior facilidade de debug, bem como a capacidade de passar valores (atributos) tanto para cima como para baixo na árvore de análise, durante a própria análise.

As especificações lexicais (tais como expressões regulares, strings, etc.) e as especificações gramaticais (usando BNF) são anotadas no mesmo ficheiro, tornando a gramática mais legível e de mais fácil manutenção.

O JavaCC contém um poderoso "*tree building preprocessor*", o JJTree.

O JavaCC oferece diversas opções para personalizar tanto o seu comportamento como o dos *parsers* gerados.

Por omissão, o JavaCC gera *parsers* LL(1). No entanto, podem haver porções da gramática que não sejam LL(1). Esta ferramenta oferece a capacidade de *lookahead* sintáctico e semântico para resolver ambiguidades nestes pontos. Desta forma, o *parser* é LL(k) apenas nesses pontos, permanecendo LL(1) nos restantes sítios para melhor desempenho.

4.2. SableCC

Este subcapítulo foi escrito com inspiração na documentação oficial do SableCC [Gagnon 98] e [SableCC web 1].

SableCC é outra ferramenta geradora de *parsers*, cuja funcionalidade é a geração de frameworks orientados a objectos para a construção de compiladores, interpretadores e outros *parsers* de texto. Em particular, as *frameworks* geradas incluem árvores de sintaxe abstracta e "*tree walkers*" intuitivas e estaticamente tipificadas. Uma importante característica desta ferramenta é a separação clara que é mantida entre código máquina gerado e código escrito pelo programador, o que conduz a um ciclo de desenvolvimento mais curto.

A mais importante funcionalidade do SableCC 3 é a capacidade de especificação de regras de transformação para criar uma AST ("Árvore de Sintaxe Abstracta") a partir das mais básicas árvores gramaticais (CST – "Árvore de Sintaxe Concreta").

Uma característica do SableCC que faz esta ferramenta diferir de outras do mesmo género é o facto de os ficheiros de especificação não conterem acções semânticas. Como alternativa, o SableCC gera uma *framework* orientada a objectos, à qual podem ser adicionadas essas acções, simplesmente definindo novas classes contendo esse tipo de código.

O SableCC gera uma Árvore de Sintaxe Abstracta estaticamente tipificada a partir da gramática presente no ficheiro de entrada, gerando de seguida uma classe abstracta representando cada não terminal da gramática, sendo essa classe abstracta estendida por uma ou mais classes concretas, cada uma representando uma regra alternativa na produção para esse não terminal. Os processamentos específicos associados a cada tipo de nó, ficam programados em métodos da respectiva classe ou subclasses.

O SableCC consiste num gerador de *parsers* LALR(1), num gerador de construtores de Árvores de Sintaxe Abstracta e num gerador de *frameworks* AST orientadas por objectos.

Exemplo

Para exemplificar o funcionamento do SableCC, é aqui apresentado um exemplo de um pequeno programa, uma simples calculadora que apenas recebe dois dígitos e efectua a sua soma. Este exemplo é retirado de um tutorial [SableCC web 2] que demonstra a facilidade com que se pode usar esta ferramenta, em particular tirando partido do Eclipse, programa usado para desenvolvimento de software.

Em primeiro lugar é apresentada a simples gramática SableCC para esta calculadora, que recebe duas expressões na forma [INT] + [INT], imprimindo de seguida o resultado.

```
/* simpleAdder.sable */
Package simpleAdder ;

Helpers
    digit = ['0' .. '9'] ;
    sp = ' ' ;
    nl = 10 ;

Tokens
    /* Definição dos Tokens */
    integer = digit+ sp*;
    plus = '+' sp*;
    semi = ';' nl?;

Productions
    /* A nossa simples gramática */
    program = [left]:integer plus [right]:integer semi;
```

Executando-se o SableCC sobre o ficheiro de especificação acima descrito, origina a criação da framework com que iremos trabalhar. São automaticamente geradas algumas *packages* e classes, como por exemplo a *package* "*analysis*", com as classes para análise numa AST, o lexer, o parser, e classes para representar a semântica da linguagem.

Note-se que se nomearam os dois elementos do tipo "integer" na produção, "left" e "right". Em baixo podemos ver que estes elementos são facilmente acedidos pelo nome atribuído.

O próximo passo é a construção de um interpretador, com vista a testar se a nossa gramática foi adicionada correctamente. Apresenta-se agora a classe **Interpreter**:

```
package simpleAdder.interpret;

import simpleAdder.node.* ;
import simpleAdder.analysis.* ;
import java.lang.System;

public class Interpreter extends DepthFirstAdapter {

   public void caseAProgram(AProgram node) {
      String lhs = node.getLeft().getText().trim();
      String rhs = node.getRight().getText().trim();
      int result = (new Integer(lhs)).intValue() + (new Integer(rhs)).intValue();
      System.out.println(lhs + "+" + rhs + "=" + result);
   }
}
```

Por fim, procedemos à criação de uma função **Main**, com o objectivo de criar a AST e de seguida invocar o interpretador definido acima:

```
import simpleAdder.interpret.Interpreter;
import simpleAdder.parser.* ;
import simpleAdder.lexer.* ;
import simpleAdder.node.* ;

import java.io.* ;

public class Main {
   public static void main(String[] args) {
      if (args.length > 0) {
         try {
            /* Form our AST */
            Lexer lexer = new Lexer (new PushbackReader(
               new FileReader(args[0]), 1024));
            Parser parser = new Parser(lexer);
            Start ast = parser.parse() ;

            /* Get our Interpreter going. */
            Interpreter interp = new Interpreter () ;
            ast.apply(interp) ;
         }
         catch (Exception e) {
            System.out.println (e) ;
         }
      } else {
         System.err.println("usage: java simpleAdder inputFile");
         System.exit(1);
      }
   }
}
```

Para testar a nossa simples linguagem, é necessário um ficheiro de texto com uma conta de somar. Ao executar-se a classe Main, com este ficheiro como argumento, o programa irá calcular, como se pretende, a soma.

4.3. Polyglot

Este subcapítulo é baseado na documentação oficial do Polyglot [Polyglot web] e [Nystromet et al. 03].

A última ferramenta a ser aqui analisada é o Polyglot, um "*compiler front end framework*" cuja funcionalidade passa pela construção de extensões da linguagem Java. Esta ferramenta é implementada como um *framework* de classes Java através de padrões de desenho com vista a promover a extensibilidade, sendo que com o seu uso as extensões da linguagem podem ser implementadas sem duplicação de código do *framework*. O Polyglot pode ser visto como uma biblioteca de classes Java, facilmente extensível através do mecanismo de herança para criar um compilador para uma linguagem que seja uma modificação do Java. O compilador de base implementa a tradução "identidade" de Java para Java; usando herança é possível estender as sintaxes concreta e abstracta, o sistema de tipos, e introduzir transformações de código.

Esta ferramenta apresenta-se como sendo bastante útil quando o programador pretende reutilizar tanto quanto possível a infra-estrutura do compilador existente, sendo que o custo da implementação das extensões da linguagem é escalável com o grau em que a linguagem difere do Java.

O Polyglot inclui o PPG, um gerador de *parsers* LALR extensível.

O compilador Polyglot está estruturado como um conjunto de passos que traduz ficheiros da linguagem fonte, uma extensão do Java, para código Java puro. Começa por analisar os ficheiros fonte criando uma AST, reescreve a AST para eliminar ambiguidades, efectua uma verificação de tipos na AST, possivelmente volta a reescrever a AST, sendo o *output* final a AST sob a forma de código fonte Java.

Com vista a permitir uma maior flexibilidade na substituição (*overriding*) do comportamento de um nó da AST, cada nó tem um apontador para um *delegate object* e uma lista (possivelmente *null*) de *extension objects*. Estes últimos objectos são úteis se quisermos

adicionar uma variável ou um método a vários nós diferentes da AST, sendo o seu propósito permitir uma extensão uniforme de vários nós da AST.

4.4. Análise comparativa face aos objectivos do trabalho

Sendo o objectivo deste trabalho a construção de um tradutor da linguagem JavaCO para Java puro, podemos concluir que qualquer das três ferramentas analisadas possui as características e funcionalidades necessárias à sua implementação, nomeadamente à analise da linguagem de entrada, sua tradução e obtenção de *output* sob a forma de código Java puro.

A ferramenta que foi escolhida foi o SableCC. Relativamente ao JavaCC tem a vantagem de gerar automaticamente a AST, algo que vai ser necessário neste projecto. A solução de usar o JavaCC com a componente JJTree também seria possível, mas concluiu-se que daria origem a uma solução um pouco mais complicada.

Relativamente ao Polyglot, a principal razão de não ter sido escolhido foi a longa curva de aprendizagem associada a esta ferramenta. Um investimento no Polyglot pareceu difícil de justificar considerando os objectivos relativamente modestos do trabalho. Teve também bastante peso na escolha efectuada o facto de se considerar que uma experiência com o SableCC seria mais útil numa perspectiva futura. Mais concretamente pelo facto desta ferramenta aceitar a gramática de qualquer linguagem, não estando comprometida com nenhuma linguagem em particular.

Sendo o SableCC a ferramenta escolhida para a implementação do tradutor, foi a única a ser ilustrada com um exemplo.

5. Descrição do Trabalho

A linguagem Java [Gosling, McGilton 96] é detentora de um sistema de tipos substancialmente estático e seguro. Se, por um lado, o sistema de tipos estático possa ser uma motivação para a escolha do Java, por outro lado ele cria restrições que se revelam contraditórias com a forma humana de pensar, quando está em causa a modelação de alguns aspectos do mundo real. Normalmente, essas situações são resolvidas em Java de uma das duas seguintes formas: contornando o sistema de tipos estático e manipulando explicitamente o tipo dos objectos (usando *casts* e *instanceof*), ou usando determinados padrões de desenho, como por exemplo o padrão do visitante [E. Gamma et al. 94]. Mas em ambos os casos a programação torna-se artificial ficando obscurecida por diversos aspectos técnicos.

5.1. Objectivo do trabalho

O objectivo deste trabalho consiste no desenho e implementação da linguagem JavaCO, uma variante da linguagem Java com suporte para um dos mecanismos mais marcantes da linguagem Eiffel.

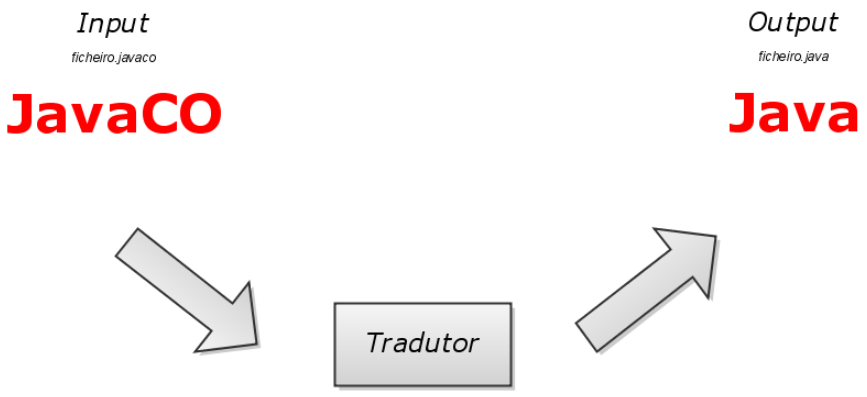
A sintaxe original do Java foi quase toda mantida, sendo a principal alteração relativa ao sistema de tipos, que é baseado em covariância, tal como acontece na linguagem Eiffel. Foi introduzida uma nova palavra reservada **covariant**, para exprimir a intenção de usar tipos covariantes nos argumentos da redefinição de métodos.

Para se poder tirar o melhor partido do mecanismo de covariância, também foi acrescentada à sintaxe do JavaCO um tipo predefinido **This**. Este novo tipo serve para representar de um modo genérico o tipo do identificador **this** dentro de cada classe. À partida **This** destinou-se apenas a ser usado como tipo de argumentos de métodos, embora numa segunda fase esta restrição tenha sido relaxada. Note-se que o tipo **This** tem de ser reinterpretado em cada subclasse, sempre que aparece em código herdado, visto que o significado dele varia, dependendo da classe que herde esse código. Já dentro de cada objecto, o significado de **This** é fixo e representa o tipo do próprio objecto.

A nova linguagem mantém compatibilidade com os programas em Java puro, ou seja, um programa em Java visto como um programa em JavaCO, mantém a sua semântica.

5.2. Tradutor

Este trabalho envolveu o desenvolvimento de um tradutor, implementado com a ajuda da ferramenta SableCC, que recebe à entrada *(input)* ficheiros da linguagem JavaCO e gera à saída *(output)* ficheiros em Java puro que implementam a semântica pretendida para a primeira linguagem.



Dada a invariância do tipo dos argumentos dos métodos em Java, um dos aspectos mais importantes da tradução é manter essa mesma invariância: à saída do tradutor, os métodos redefinidos de forma covariante têm de ter todos a mesma assinatura. Desta forma, os métodos em questão não serão considerados *overloaded*, mas sim *overriden*. Recorde-se que em Java, a assinatura de um método consiste no nome e na sequência dos tipos dos argumentos; não inclui o tipo do resultado nem quaisquer modificadores. Além disso, o Java já suporta covariância nos resultados dos métodos.

O processo de tradução divide-se em três fases principais: na primeira fase, é analisada a AST, recolhida informação e são efectuadas algumas validações, como por exemplo a ocorrência de modificadores **covariant** ou tipos **This** em sítios não permitidos; na segunda fase completa-se a informação sobre as classes com base nos dados recolhidos na etapa anterior e efectuam-se as restantes validações; na terceira fase, procede-se à alteração da AST, com vista à obtenção do código Java final, que será então o *output* do tradutor.

Estas três fases serão referenciadas ao longo deste documento como fase 1, 2 e 3, respectivamente.

Execução do tradutor

Para executar o tradutor, o programador deverá situar-se na directoria onde se encontra o ficheiro JavaCO.jar e o ficheiro de entrada com extensão “javaco”, e ai digitar o comando:

> java –jar JavaCO.jar *ficheiro_de_entrada.javaco*

5.3. Abordagem e princípios orientadores deste trabalho

Para começar, convém tornar claro que a implementação do mecanismo de covariância usa tipificação dinâmica de forma substancial. No caso dos métodos covariantes é imitada a abordagem do Eiffel original em que a violação dos tipos dos argumentos dos métodos

covariantes não é detectada em tempo de compilação mas sim em tempo de execução (as implementações mais recentes da linguagem Eiffel já não se comportam assim).

Outro princípio importante que seguimos é que o tradutor não repete a análise de tipos da linguagem Java. O tradutor de JavaCO deixa para o compilador de Java essas validações, tendo sido necessário desenvolver estratégias para lidar com programas contendo erros de Java. Por exemplo, é necessário lidar com classes usadas mas não definidas e lidar com classes homónimas. Este aspecto será explicado com algum detalhe nas secções dedicadas à implementação. Convém dizer que não estão em causa erros sintácticos independentes do contexto, pois a primeira fase do tradutor, implementada pelo gerador de *parsers* SableCC, já se encarrega de detectar esses erros durante a geração da AST.

Apesar de se recorrer a tipificação dinâmica, há muitos aspectos que podem ser validados estaticamente. Neste trabalho procuram-se identificar todas essas situações pois é sempre preferível detectar os erros em tempo de compilação do que em tempo de execução. Algumas dessas validações estáticas são feitas em tempo de tradução e podem gerar mensagens de erro. Eis alguns exemplos:

- O uso do modificador **covariant** é validado em todas as suas utilizações, assim como em todas as suas omissões incorrectas.
- A covariância dos argumentos dos métodos covariantes é validada.
- A proibição de fazer *overloading* de métodos covariantes é validada.

Outras validações estáticas são feitas indirectamente através da geração de código Java preparado para provocar determinadas mensagens de erro, caso haja problemas com o código JavaCO original. Por outras palavras, gera-se um determinado código Java, contando com as validações estáticas que o compilador de Java irá fazer.

O último princípio é que qualquer violação do sistema de tipos cuja validação tenha de ser feita dinamicamente, deve ser detectada o mais cedo possível, no momento exacto em que a violação ocorre. Desta forma não haverá código que seja executado numa situação de inconsistência ao nível dos tipos. O respeito por este princípio também servirá de base à

nossa argumentação que o sistema de tipos da nova linguagem é seguro, ou seja, rejeita operações que não respeitem os tipos de dados.

Muitos destes princípios denotam uma atitude de rigor e de preocupação com a segurança do sistema de tipos da linguagem. Contudo deve ser dito que neste trabalho não se formaliza o sistema de tipos nem há qualquer tentativa de fazer uma demonstração da sua segurança.

Para compensar um pouco a situação deve ser dito que foi extraída alguma orientação duma linguagem existente e completamente formalizada, chamada PolyTOIL [K.B. Bruce et al. 03]. Esta linguagem tem uma construção *MyType* covariante bastante semelhante ao **This** do JavaCO e o estudo das suas regras de tipo ajudou a não esquecer nenhuma situação de potencial insegurança.

5.4. Covariância nos argumentos dos métodos

Com o intuito de dirigir a programação em JavaCO na direcção de um pensamento mais humano e mais aproximado ao mundo real, foi implementada covariância nos argumentos dos métodos redefinidos por subclasses. Esta é a mesma filosofia da linguagem Eiffel.

Com o intuito de proporcionar uma escrita clara na programação nesta nova linguagem, foi introduzida uma nova palavra reservada **covariant**, a empregar cada vez que se define ou redefine um método, no qual o programador resolveu permitir covariância nos argumentos. O uso da palavra **covariant** também implica que se proíbe sobrecarga (*overloading*) do nome do método envolvido. Todos os métodos que não tenham o modificador **covariant** serão tratados como métodos Java normais, podendo inclusivamente ser *overloaded*.

JavaCO

```
class Animal {
  public covariant void comer(Alimento al){
    ...
  }
}

class Vaca extends Animal {
  public covariant void comer(Erva e){
    ...
  }
}
```

5.5. Introdução da palavra-chave This

Faz parte da linguagem JavaCO a palavra-chave (*keyword*) **This**, que pode ser usada no tipo dos argumentos dos métodos, e que representa de um modo genérico o tipo do identificador **this** dentro de cada classe. Os parâmetros de tipo **This** são *read-only*.

A introdução do **This** visa explorar ao máximo o potencial da covariância. Na prática, a adição deste mecanismo tem o efeito de tornar o mecanismo de herança mais expressivo. A linguagem Eiffel tem um mecanismo equivalente, usando a sintaxe **like Current** [Meyer 06].

O termo **This** tem um significado flexível, no sentido em que, para além de representar o tipo da instância da classe em que aparece escrito, tem de ser reinterpretado nas subclasses, cada vez que ocorre em código herdado (o que acontece com **this**, que representa uma entidade de carácter variável). O significado de **This** vai mudando automaticamente pela hierarquia abaixo, quando o método é herdado pelas subclasses.

5.6. Uso estendido da palavra-chave This

A utilização da palavra **This** nos argumentos dos métodos, que foi descrita na secção anterior, é a mais útil para o programador. Contudo, resolvemos liberalizar o uso dessa palavra, permitindo que ela ocorra no resultado das funções e na definição de variáveis, tanto

locais como de instância. Veremos mais adiante que, em resultado da circunstância de **This** não representar um tipo concreto, esta liberalização obriga o programador a usar determinados esquemas de programação que, numa primeira fase, poderão ser pouco intuitivos.

Esta liberalização do uso da palavra **This** vai também na linha da linguagem Eiffel, e das possibilidades do uso do tipo **like Current**.

5.7. Tipos genéricos e parâmetros de tipo

Uma das conclusões deste trabalho é que não é possível estender de forma satisfatória a noção de covariância a tipos genéricos e a parâmetros de tipo. Isto acontece porque implementámos a covariância pela via dos tipos dinâmicos, mas o Java, como é sabido, remove toda a informação de tipo associada aos parâmetros de tipo, num processo que se chama *type erasure*.

Relativamente a tipos genéricos e parâmetros de tipo, a linguagem JavaCO acaba por herdar todas as limitações já contidas na linguagem Java.

5.8. Implementação

A implementação do tradutor foi feita recorrendo à ferramenta SableCC, que nos vai disponibilizar a Árvore de Sintaxe Abstracta dos programas, a qual iremos percorrer diversas vezes através dos *walkers* também ao nosso dispor. Esta ferramenta gera parsers escritos em Java, o que no caso deste trabalho foi uma vantagem visto que o objectivo é processar uma linguagem da família do Java.

A tradução de JavaCO para Java puro é feita recorrendo a quatro *walkers* principais. No entanto, para efeitos de validação do uso estendido da palavra-chave **This**, é necessário utilizar três *walkers* extra.

Dos quatro *walkers* que funcionam como base para a tradução, os dois que primeiro visitam a árvore têm o intuito de recolher informação, na sua maioria relacionada com as classes e métodos do programa de entrada. Depois de algumas validações efectuadas ao nível dos dados recolhidos, os outros dois *walkers* encarregam-se de operar a tradução propriamente dita.

Os outros três *walkers* têm como objectivo a criação de classes auxiliares que servem para a validação do uso estendido de **This**. As vantagens e modo de funcionamento de todos os *walkers* serão descritos oportunamente em capítulos posteriores.

Foram criadas diversas classes Java, na sua maioria contendo estruturas de dados para armazenar a informação recolhida. Uma das mais importantes, *Globals*, alberga duas estruturas de dados estáticas, e portanto partilhadas por todo o programa. A primeira estrutura, chamada *classesAndInterfaces*, guarda informação relativa às classes e interfaces; a segunda, chamada *methods*, guarda informação sobre todos os métodos encontrados. Nesta classe foram também depositados diversos métodos, todos estáticos e que fornecem funcionalidades necessárias em diversas das etapas de todo o processo de tradução.

Das restantes classes, importa destacar as seguintes: *XClassOrInterface* (informação relacionada com classes e interfaces), *XMethod* (dados sobre métodos), *MethodHeader* (detalhes das assinaturas dos métodos) e *BestTypes* (envolvida no algoritmo de escolha dos melhores tipos para substituição).

Por fim, outra classe não menos importante é a classe *Error*, apenas com uma variável estática e métodos estáticos. A variável guarda uma estrutura de dados onde são armazenados todos os erros gerados durante as validações efectuadas pelo tradutor. O tradutor emite mensagens de erro quase só relacionadas com violações envolvendo o mecanismo de covariância. Por exemplo, desrespeito das regras de subtipo no cabeçalho dos métodos covariantes, a omissão inválida do modificador **covariant** num dado método, ocorrências inválidas do tipo **This**. É relevante frisar que se tiverem sido detectados erros durante a fase de validação, serão apenas emitidas as mensagens de erro correspondentes, não sendo efectivada a tradução de JavaCO para Java.

6. Covariância

6.1. Covariância em JavaCO

Apresenta-se de seguida um exemplo escrito em JavaCO, cujos detalhes serão evidenciados ao longo do capítulo corrente.

Comecemos por analisar um caso em que a contravariância nos restringe relativamente a uma expressividade natural. Suponhamos que temos um tipo `Animal`, contendo os métodos `comer(Alimento al)` e `reprodução(Animal an)`. Consideremos também um tipo `Vaca`, subtipo de `Animal`.

JavaCO

```
class Animal {

        covariant public void comer(Alimento al){
                ...
        }

        covariant public void reprodução(Animal an){
                ...
        }

}

class Vaca extends Animal {

        covariant public void comer(Erva e) {
                ...
        }

        covariant public void reprodução(Vaca v) {
                ...
        }

}
```

Assuma-se também a existência de um tipo `Erva`, subtipo de `Alimento`, designando o único tipo de alimentação passível de ser ingerida por um animal do tipo `Vaca`.

De forma a respeitar o sistema de tipos do Java, é-nos impossível redefinir este método na classe `Vaca`, introduzindo como parâmetro esperado um tipo mais restrito que o que se encontra na declaração do método na classe `Animal` (em Java esse seria considerado um método distinto, não uma redefinição, em virtude de sobrecarga).

No entanto, para frisar que uma vaca é um animal que não se alimenta de qualquer variedade de alimentos, seria importante redefinir o método `comer`, de maneira a que este recebesse unicamente `Erva` como alimento.

Da mesma forma, é útil que nas subclasses de `Animal` se permita a redefinição do método `reprodução`, em que o argumento será sempre do mesmo tipo que a classe em que é redefinido.

Este exemplo serve para ilustrar a situação frequente em que num contexto mais específico se sente a necessidade de usar tipos mais especializados. Este género de situação surge frequentemente quando lidamos com uma hierarquia de classes, sejam animais, veículos, formas, etc. Torna-se especialmente notória quando estamos perante duas ou mais hierarquias interdependentes ou quando precisamos de definir métodos binários (métodos em que o argumento tem o mesmo tipo do receptor).

Neste capítulo consideramos apenas tipos não genéricos. As implicações da covariância nos tipos genéricos será discutida noutro capítulo, mais adiante.

6.2. Regra formal

Como já foi abordado no Capítulo dos Sistemas de Tipos, o que garante segurança a um Sistema de Tipos estático é a contravariância dos argumentos dos métodos e a covariância dos resultados.

A regra do Java é estaticamente segura e compatível com a regra formal, visto que usa invariância para os argumentos e covariância para os resultados. A covariância dos resultados foi introduzida em Java na versão 5.0.

Considerando que o Java já suporta covariância no tipo dos resultados, apenas temos de nos preocupar com a introdução de covariância nos argumentos.

A regra formal da covariância, que vamos usar, é a seguinte:

$$\frac{A <: A' \qquad B <: B'}{A \rightarrow B \; <: \; A' \rightarrow B'}$$

Esta regra não dá garantias estáticas de segurança, pelo que como já foi dito, usaremos tipificação dinâmica para obter essas garantias.

6.3. Covariância noutras linguagens

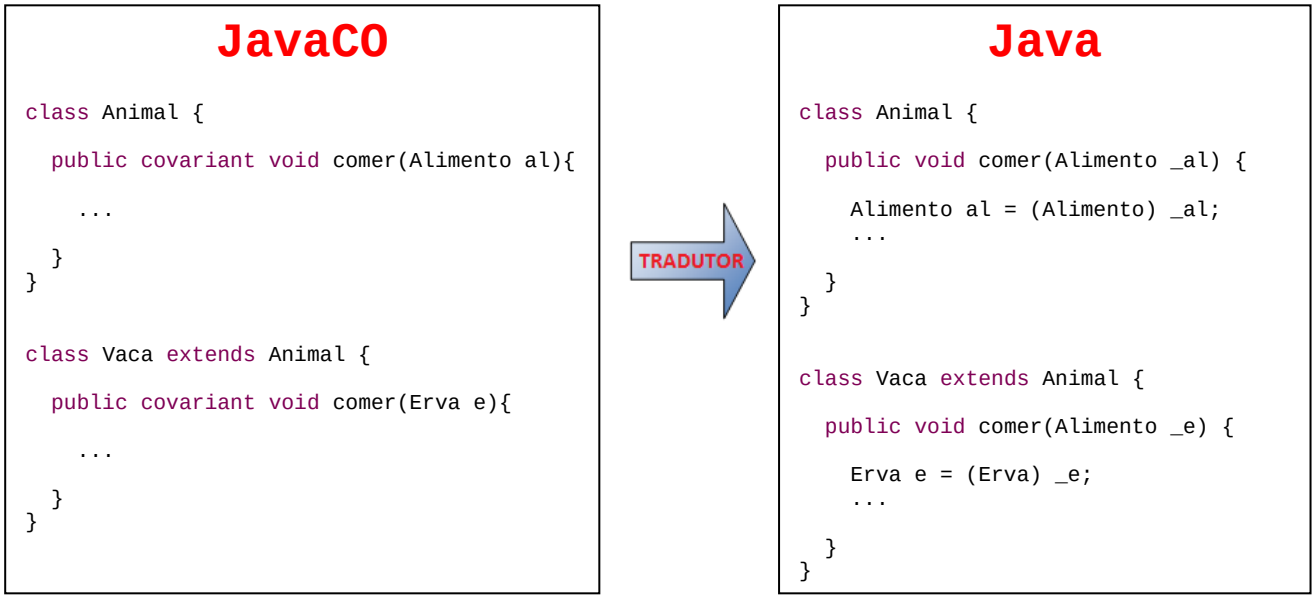
Tal como acontece em Java, também outras linguagens de programação permitem covariância no tipo de retorno de métodos, como por exemplo as linguagens C++, Sather e Eiffel.

A linguagem Sather [Sather web] [Omohundro, Lim 92] tem a particularidade de, relativamente aos tipos dos argumentos, suportar covariância para parâmetros de saída e contravariância para parâmetros normais, de entrada.

O Eiffel é a linguagem que permite a covariância dos argumentos, e na qual baseamos o nosso pensamento para autorizar que o mesmo aconteça na linguagem JavaCO.

6.4. Tradutor

De seguida é demonstrado, recorrendo a código Java, a forma como o tradutor implementado lida com a covariância nos tipos dos argumentos.



Pretende-se permitir o uso de um subtipo de `Alimento` no argumento da redefinição do método `comer` numa subclasse de `Animal`. Estamos assim a especializar o tipo do argumento, restringindo a variedade de objectos que o método pode receber.

Para se manter a consistência com o que acontece em Java relativamente ao uso de modificadores, **covariant** deve ser escrito repetidamente, sempre que se define ou redefine um método covariante numa classe, ou se declara ou redefine um método covariante numa interface. Este modificador tem de aparecer sempre na zona dos modificadores em Java – antes do tipo de retorno do método.

O tradutor verifica – na segunda fase da tradução – se o tipo de cada argumento é compatível com aquele que foi declarado na(s) superclasse(s), validando desta forma, estaticamente, a covariância dos parâmetros para a assinatura de cada método. Caso todas as validações sejam correctas, todo o corpo do método pode ser transcrito na sua totalidade para o *output* do

tradutor. Mas a validação estática não é suficiente, e o tradutor insere validações dinâmicas de tipo para verificar em tempo de execução se os argumentos do método têm os tipos dinâmicos esperados.

6.5. Implementação

6.5.1. Armazenamento dos tipos, suas hierarquias e métodos existentes

Tendo em conta os nossos objectivos há a necessidade de, em várias etapas da tradução, ter conhecimento, tanto das relações de subtipo existentes no programa em análise, como dos métodos existentes.

Com vista a satisfazer esta necessidade, usam-se duas *Hashtables*, uma para guardar a informação relativa a todos os tipos encontrados na AST do programa, outra com o intuito de armazenar alguns dados importantes referentes aos métodos. O programa de entrada é analisado, tal como as classes e interfaces existentes, das quais se extrai a informação relevante para a tradução a efectuar, por exemplo a relação de herança entre essas entidades.

As duas estruturas de dados referidas são estáticas, o que significa que existe apenas uma instância de cada *Hashtable* durante todo o processo de tradução. Ambas são variáveis da classe *Globals.java*, classe esta cujo conteúdo se aprofundará mais adiante.

As duas estruturas definem-se assim:

```
Hashtable<String,XClassOrInterface> classesAndInterfaces
```

Hashtable cuja *key* é uma *String* que denota o nome completo e único de uma classe, e *value* um objecto *XClassOrInterface*, onde é guardada a informação sobre cada classe ou interface usada;

```
Hashtable<Node,XMethod> methods
```

Hashtable cuja *key* é um objecto do tipo *Node*, com o próprio nó do método, e *value* um objecto *XMethod*, onde se guarda toda a informação sobre cada método usado.

6.5.2. Implementação da regra de escopo estático

A linguagem JavaCO, tal como a linguagem Java, usa escopo estático (também conhecido como escopo lexical) como regra de resolução de nomes não locais. Começamos por apresentar a forma como o tradutor implementa a regra, em particular a forma como lida com o imbricamento de classes e métodos.

6.5.2.1. Obtenção do nome completo de uma classe ou interface

No contexto deste trabalho, chamamos **nome completo** de uma classe/interface ao seu nome completamente qualificado, composto pelos nomes das classes, interfaces e métodos circundantes separados por pontos, e reflecte o imbricamento dessas entidades. Faz ainda parte do nome completo, como prefixo, o *package* a que a entidade pertence. Cada nome completo terá também pontos nas suas extremidades. Assim, se definirmos uma classe *B* dentro de uma classe *A* no *package* por omissão, o nome completo da classe *B* será *".A.B."*.

Quando na linguagem Java se define uma classe *B* dentro de uma classe *A*, geralmente não a declaramos como *A.B*, que é na realidade o seu verdadeiro nome para o compilador de Java. Normalmente opta-se por declará-la apenas como classe *B*. Denominamos este nome simplificado por **nome básico** de uma classe/interface.

Se considerarmos a classe de biblioteca do Java *java.lang.String*, designamos como seu nome completo *".java.lang.String."*. O seu nome básico é *String*.

Nos dois primeiros *walkers* da implementação, a cada classe ou interface existente no programa JavaCO de entrada, é atribuído um nome completo. Cada um destes nomes é único e representa uma classe/interface única.

6.5.2.2. Uso de StringBuffer para gerar nomes completos

Nos dois primeiros *walkers*, os nomes completos são construídos com base numa variável do tipo *StringBuffer*, que vai sendo alterada à medida que se entra e sai em classes, interfaces e métodos.

Manter esta variável actualizada durante todo o percurso da AST é a forma de, em qualquer altura do percurso, termos informação sobre o ambiente lexical envolvente. Assim conseguimos construir o nome completo de uma qualquer classe a partir do seu nome básico.

A variável referida é inicializada com um carácter ponto (*“.”*). Se entrarmos numa classe *Carro*, a variável passa a *“.Carro.”*. Se de seguida entrarmos na classe *Roda*, a variável tomará o valor *“.Carro.Roda.”*. Ao sair desta última classe, a variável volta a *“.Carro.”*.

6.5.2.3. Particularidade na entrada em métodos

Sendo que uma classe ou interface pode em Java ser definida dentro de um método, também os nomes dos métodos têm de entrar nos nomes completos das classes e interfaces. Desta forma, também os nomes dos métodos são adicionados e retirados do *StringBuffer*, à entrada e saída dos métodos, respectivamente.

Para se obter uma clara distinção entre classes/interfaces e métodos, à entrada de um método o *StringBuffer* é estendido com o seu nome precedido de um símbolo cardinal (*“#”*).

6.5.2.4. Semelhanças com filosofia da linguagem Java

Esta forma de construção do nome completo de uma classe vem ao encontro do que se passa em Java, onde também se separam os nomes das classes por pontos, omitindo-se no entanto os pontos no início e no fim. A decisão da adição destes dois permite simplificar muito a operação que testa se um nome completo faz parte de outro nome completo. Basta testar directamente se um nome é *substring* de outro, sem ser preciso lidar explicitamente com os pontos e a estrutura linear por eles definida.

Por exemplo, o nome *Cidade*, sem os dois pontos nas extremidades é parte do nome *"Z.UniverCidade"*, mas o nome *".Cidade."* já não é parte de *".Z.UniverCidade."*.

6.5.2.5. Resolução de nomes

Os únicos nomes que o tradutor precisa de resolver são os nomes dos tipos dos parâmetros dos métodos. Portanto, o que está em causa é o seguinte: dado um nome de tipo, que pode ou não ter pontos pelo meio, descobrir qual a classe ou interface que lhe corresponde.

Imaginemos que nos surge um parâmetro de um método cujo tipo é *"C.D"*. Imaginemos também que o imbricamento de classes corrente é *".A.B.C."*, ou seja, o contexto sintáctico corrente é o do interior da classe *"A.B.C"*.

A descoberta da definição da classe *"C.D"* que está a ser usada a partir do ponto corrente é feita pelo método de tentativa e erro. Consideram-se sucessivamente todos os possíveis pontos da definição da classe *"C.D"*, até se descobrir que um deles corresponde a uma classe existente. A pesquisa é feita de dentro para fora, considerando os vários níveis de aninhamento. Eis a sucessão de tentativas efectuadas para o nosso exemplo:

.A.B.C.C.D.

.A.B.C.D.

.A.C.D.

.C.D.

6.5.2.6. Como lidar com um programa errado que contenha classes homónimas?

Porque nada nos garante que o programa JavaCO de entrada se encontra correcto, o nosso mecanismo de nomeação dispõe de uma funcionalidade que, no seguimento da filosofia adoptada de deixar os erros de Java para o compilador de Java, nos permite lidar com classes duplicadas.

Se aparecerem múltiplas definições de uma classe, por exemplo, *".Transporte.Carro."*, a classe que ocorre em primeiro lugar fica com esse nome. Se nos aparecer outra classe/interface com o mesmo nome, adicionamos ao nome um acento circunflexo ("^"). Assim, o nome completo desta classe/interface será *".Transporte.Carro^."*. Se mais alguma classe ou interface surgir nestas condições, o seu nome completo incluirá dois acentos circunflexos, e por ai adiante.

A consequência desta nomeação é que, quando se procurar pela classe *".Transporte.Carro."*, será sempre devolvida a primeira, sendo que as restantes, às quais foram atribuídos nomes "falsos", nunca poderão ser acedidas. Apenas se optou por manter informação sobre as mesmas para permitir que o seu conteúdo possa ser validado relativamente à covariância.

6.5.2.7. Uso de Stacks para lidar com imbricamento de classes e métodos

Nos dois primeiros *walkers* temos também um *Stack* de *String*, que nos permite manter, em formato de pilha, os nomes das classes e das interfaces em que estamos em cada momento. Existe também um *Stack* de *Node* para manter os nós dos métodos com a mesma finalidade.

Estas pilhas são muito úteis, permitindo conhecer constantemente qual o contexto sintáctico envolvente. Irão assim permitir a interpretação da AST, que por vezes é tudo menos trivial devido ao seu conteúdo complexo.

Um exemplo de utilidade destas pilhas é quando se itera sobre a AST e se encontra um nó de um parâmetro de um método. Se acedermos ao topo da pilha de métodos, obtemos o método corrente, ao qual pertence o parâmetro em questão.

6.5.3. Uso de walkers

A implementação do tradutor foi feita recorrendo à ferramenta SableCC, que nos disponibiliza a Árvore de Sintaxe Abstracta dos programas, a qual percorremos diversas vezes através dos *walkers* que podemos definir. Estes *walkers* são particularmente profícuos, permitindo-nos, entre outras funcionalidades, ler e alterar a informação de cada nó da árvore, tanto à entrada como à saída desse nó, ao longo da iteração por toda a AST.

A tradução de JavaCO para Java puro é feita recorrendo a quatro *walkers* **principais** (mais tarde serão usados outros *walkers*, cujas funções estão relacionadas com a validação do uso do tipo **This** na criação de variáveis e no retorno dos métodos). Os dois primeiros *walkers* tratam da análise da árvore do programa, recolha de informação e detecção de alguns erros (fase 1), enquanto os últimos dois *walkers*, após diversas verificações intermédias (fase 2), tratam da transformação da árvore (fase 3), da qual irá resultar um programa em Java puro.

A necessidade do uso de dois *walkers* distintos para a recolha de informação deve-se à necessidade de, em determinadas situações da passagem pela árvore, requerer um já total conhecimento do que está na árvore, como por exemplo a noção de todas as classes e interfaces que tem o programa que estamos a tratar.

Por outro lado, a necessidade de dois *walkers* para a transformação da árvore é justificada por uma mera razão técnica: a falta de controlo sobre a ordem pela qual o *walker* percorre os nós. Na realidade, quase todo o processo de tradução pode ser implementado num único *walker*, o terceiro. Infelizmente, neste *walker* passa-se pelos nós que representam as assinaturas dos

métodos demasiado cedo, pelo que o processamento destes nós é efectuado num quarto *walker*. Se as assinaturas fossem modificadas directamente no terceiro *walker*, quando se chegasse à inserção da criação da variável auxiliar que foi apresentada no subcapítulo 6.4., o tipo dessa variável já não estaria disponível.

Desta forma, a solução que garante a robustez e exactidão do código Java gerado passa por alterar estes nós apenas no fim da transformação, no quarto *walker*. Evita-se assim a perda de informação e uso de dados errados, aquando da geração dos novos nós, que acontecerá no terceiro *walker*.

Nas duas subsecções seguintes descrevem-se os dois primeiros *walkers* com algum pormenor. Como se pode ver nem todas as tarefas estão relacionadas com a implementação da regra de escopo estático. Quanto ao terceiro e quarto *walkers*, são eles que efectuam a tradução para Java puro através da transformação da árvore. A sua descrição é feita um pouco mais adiante, na secção 6.5.4., designada "Tradução".

6.5.3.1. Acções importantes realizadas no primeiro walker

- São adicionadas classes identificadoras de parâmetros de tipo, tanto de classes como de interfaces: Se uma classe cujo nome completo é *".Carro."* tiver sido declarada com o parâmetro de tipo *T*, será adicionada uma nova *XClasseOrInterface* com o nome *".Carro.T."*. Este objecto será assinalado como sendo representativo de um parâmetro de tipo;
- É criado o objecto *XClassOrInterface* para cada classe/interface encontrada e introduzido na *Hashtable* de classes e interfaces da classe *Globals*;
- É criado o objecto *XMethod* para cada método e introduzido na *Hashtable* de métodos da classe *Globals*;
- Para cada método, o seu identificador é adicionado ao objecto *MethodHeader* que lhe está associado e que representa a sua assinatura;
- É validado o contexto sintáctico das ocorrências do modificador **covariant** no programa;

- Se um método for privado, introduz-se essa informação no *XMethod* respectivo.

6.5.3.2. Acções importantes realizadas no segundo walker

- Quando se sai de uma classe/interface, assinala-se na respectiva *XClassOrInterface* que a mesma já foi visitada no segundo *walker*. Esta anotação será útil caso o programa original se encontre errado e existam, por exemplo, duas classes com o mesmo nome básico *"C"*. Desta forma, ao encontrar a segunda dessas classes o *walker* saberá que a primeira já foi visitada. Consequentemente os métodos interiores à classe serão assinalados como pertencentes à classe correcta, contornando-se assim a possibilidade de serem assinalados como pertencentes à *XClassOrInterface* da classe já visitada;
- Caso uma classe herde de uma superclasse, esse facto é assinalado acrescentando-se à *XClassOrInterface* respectiva um objecto do tipo *Parent*, do qual fazem parte duas variáveis: uma *String* para o nome completo da superclasse e uma referência para a *XClassOrInterface* que representa a superclasse;
- Caso uma classe implemente (ou uma interface estenda) uma ou mais interfaces, o tratamento referido anteriormente aplicar-se-á igualmente. Cada interface herdada dará origem a um objecto *Parent*, que se associará à classe ou interface "filha". Desta forma todas as *XClassOrInterface* ficarão associadas a todos os seus "pais";
- Quando se entra num parâmetro tipo referência na assinatura de um método, obtém-se o nome completo da classe que representa na nossa hierarquia de classes. No objecto *MethodHeader* correspondente ao método, guardamos o nome obtido.

  Caso se trate de um método covariante, ao *MethodHeader* é também adicionada uma referência para o nó do parâmetro. Como já foi dito, a cada cabeçalho de um método covariante está também associado um objecto *BestTypes*, que irá conter a informação necessária para a posterior escolha dos melhores tipos a utilizar na substituição dos argumentos covariantes, aquando da geração de código Java. A este objecto é também adicionado o nome completo da classe que representa o tipo deste argumento.

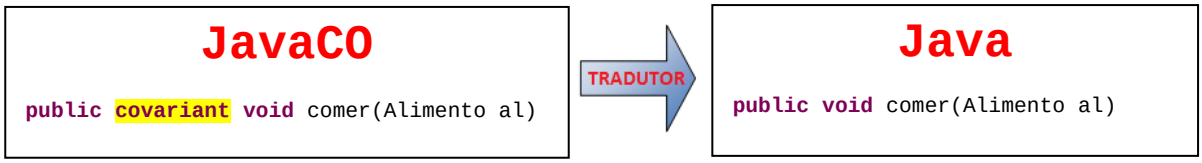
- Quando se entra num parâmetro primitivo na assinatura de um método, é também adicionada ao *MethodHeader* uma referência para o nó do parâmetro. Esta informação é necessária para verificar a compatibilidade entre dois parâmetros primitivos (que só serão compatíveis se forem iguais).

Adicionalmente, o segundo *walker* contém um método bastante importante que nos devolve o nome completo de uma classe a partir do seu nome básico e do estado corrente do *StringBuffer*. O método é o *getRealClassOrInterfaceName(String name, StringBuffer sb).*

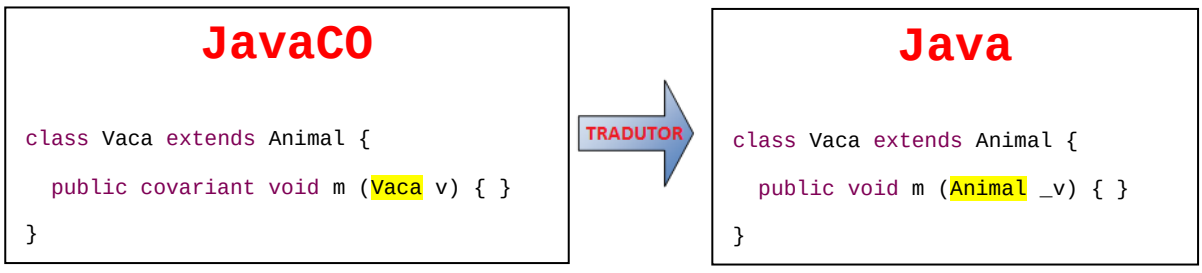
6.5.4. Tradução

A tradução é efectuada nos *walkers* 3 e 4. Eis os procedimentos constantes em qualquer tradução de um método covariante:

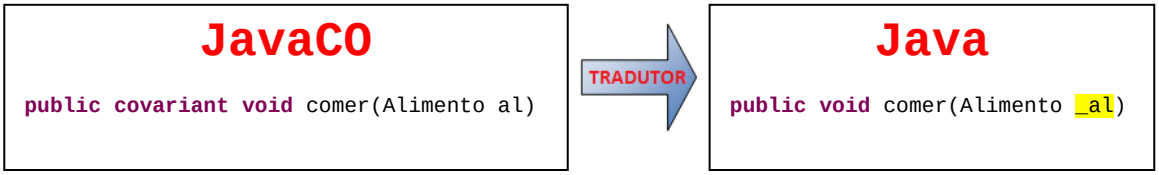
- Retirar o modificador **covariant**;



- Trocar, nas assinaturas dos métodos covariantes, cada argumento covariante pelo melhor tipo encontrado pelo algoritmo descrito no subcapítulo 6.5.6., para esse parâmetro;

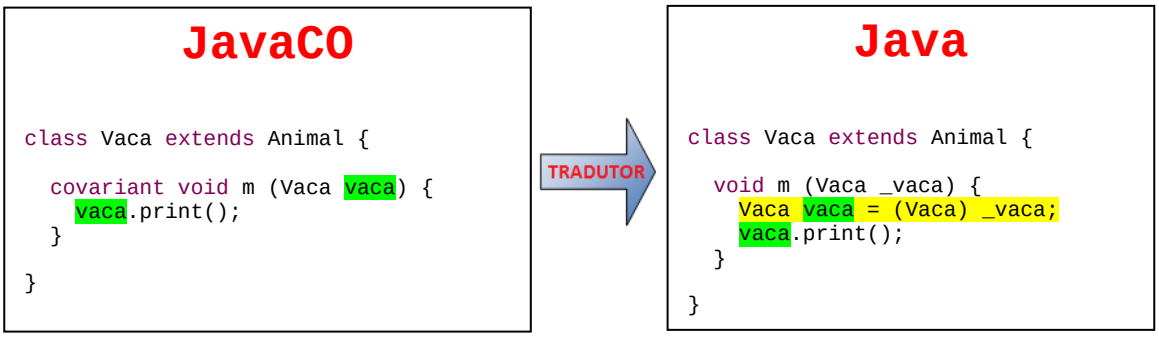


- Adicionar, como prefixo dos identificadores dos parâmetros covariantes, um carácter *underscore* " _";



- Para cada um dos argumentos covariantes presentes no cabeçalho de um método, criar uma variável auxiliar com o mesmo tipo e identificador de cada um desses parâmetros. Esta variável é uma cópia (feita através de um *cast* para o mesmo tipo) do parâmetro correspondente presente na tradução do cabeçalho.

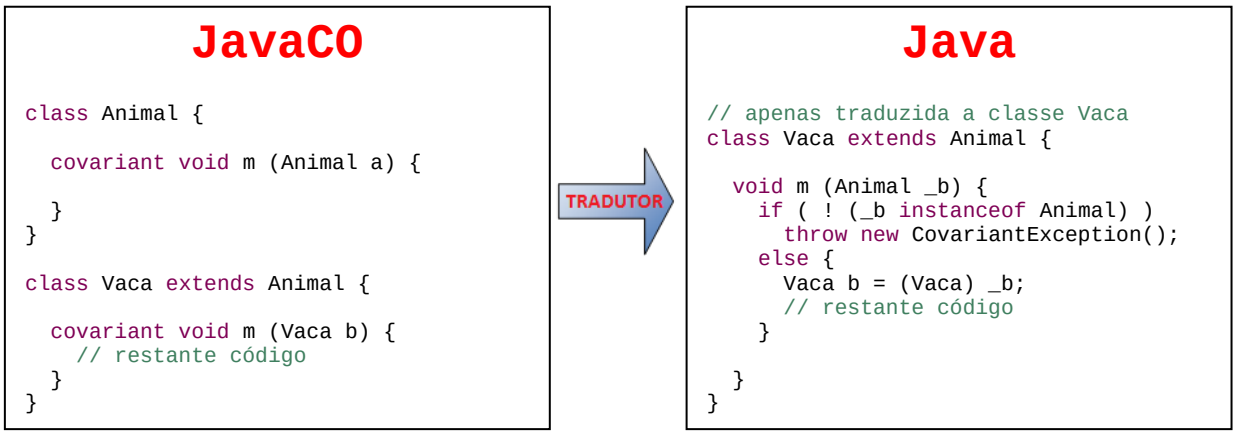
  As definições de variáveis descritas são colocadas no ínicio do código existente no corpo do método, que de resto será mantido na sua totalidade. Futuras referências ao parâmetro em questão no corpo do método usarão agora a variável declarada neste ponto. Como se pode constatar no exemplo seguinte, quando no código original do método se referencia o parâmetro *vaca*, actuamos sobre a nova variável, obtendo o comportamento pretendido para o programa original.



No final desta dissertação foram incluídos, em anexo, excertos de código da implementação relativos aos aspectos da tradução em cima referidos.

6.5.5. Não uso de instanceof

De seguida é transcrito o exemplo de uma tradução, numa versão mais preliminar do tradutor.



Como se pode verificar, nesta primeira implementação fazia-se a verificação dinâmica do tipo dos argumentos covariantes através do uso de *if* e *instanceof*, com o intuito de lançar uma excepção nos casos em que o tipo do parâmetro não era o esperado.

Esta verificação foi posteriormente retirada porque se descobriu que não era necessária: a operação de *cast* já faz a validação pretendida gerando a excepção apropriada.

6.5.6. Descrição do algoritmo para a escolha dos melhores tipos

Uma das principais particularidades deste trabalho é a obtenção, no código Java de saída, de assinaturas iguais nos métodos covariantes relacionados por redefinição, de forma a manter a invariância requerida pela linguagem Java. Põe-se então a questão de qual o melhor tipo a usar na transformação final, para cada argumento de cada método covariante.

Um dos desafios desta dissertação foi precisamente desenvolver um algoritmo que permitisse a obtenção do tipo mais adequado para substituição, em cada caso. Mais adequado, significa aqui o tipo mais concreto possível, que irá potenciar alguma detecção estática de erros nos

tipos dos argumentos do programa Java gerado. De seguida são detalhadamente descritas três alternativas possíveis para a resolução deste problema.

1) *Object sempre*

A classe *Object* é a raiz da hierarquia de classes. Quer isto dizer que qualquer classe definida em Java tem *Object* como superclasse. Assim sendo, a mais trivial opção seria optar por substituir todos os tipos de argumentos covariantes por *Object* na transformação final da AST. Este é um caso extremo no qual não se permite ao Java que efectue qualquer validação estática nas chamadas de métodos covariantes.

2) *Melhor tipo encontrado, ou Object*

A covariância dos argumentos em métodos covariantes e suas redefinições irá consistir, normalmente, na especialização dos tipos dos seus argumentos à medida que se desce na hierarquia. Assumamos a existência de um método covariante com diversas redefinições. Este método possui um só argumento covariante, cujo tipo varia nas redefinições. Após a verificação de compatibilidade dos tipos deste argumento em todos estes métodos relacionados, temos que escolher um desses tipos para a substituição em todas as assinaturas. Esta solução remete-nos para a selecção, de entre os tipos **encontrados** anteriormente, de um que seja ascendente de todos os outros na hierarquia de classes, caso exista. Se não tivermos a sorte de tal tipo existir, a escolha será o tipo *Object*.

3) *Escolha do supremo, ou Object*

Retomemos o exemplo da alínea anterior, em que obtivemos os tipos **encontrados** num método covariante e suas redefinições. Numa hierarquia complexa, pode não ser simples achar o tipo ideal para substituição, como o caso em que nenhum dos tipos encontrados é ascendente de todos os outros. É aqui que aparece o nosso conceito de tipo supremo, que

denota o mais estrito ascendente de todos os tipos encontrados. A primeira fase do algoritmo consiste, para cada tipo **encontrado**, na geração de uma lista contendo **todos** os seus ascendentes na hierarquia geral de classes, incluindo a classe *Object*. Posteriormente, se fizermos a intersecção de todas as listas anteriormente referidas iremos obter uma lista com todos os ascendentes que são comuns a todos os tipos encontrados. Note-se que todos estes tipos poderão ser usados para substituição do argumento em causa na assinatura do método e suas redefinições.

A lista nunca se encontra vazia porque inclui sempre o tipo *Object*. Se existir um elemento que seja subtipo de todos os outros, isso significa que existe um tipo supremo, e é ele o escolhido. Caso não exista supremo, a nossa alternativa é escolher um tipo ao acaso da lista, de preferência que não seja o tipo *Object*.

Por ser a solução mais aliciante ao proporcionar o resultado mais adequado, foi escolhida a alternativa 3 para a resolução do problema da selecção do melhor tipo para a substituição. Para isso, um objecto de tipo *BestTypes* foi adicionado ao *MethodHeader* de cada método, para guardar a informação necessária sobre os melhores tipos a usar na produção de código Java. Na fase de construção das listas de métodos herdados por cada classe, o conteúdo destes objectos vai sendo aumentado por acumulação. As adições são efectuadas quando se verifica a existência de compatibilidade entre dois métodos covariantes e consistem em guardar as listas dos ascendentes dos argumentos covariantes de ambos os métodos num dos objectos *BestTypes*, que acabará por ser partilhado pelos métodos após este tratamento de informação.

6.5.7. Particularidades na substituição de tipos

O algoritmo anteriormente exposto descobre qual o tipo mais adequado para a substituição de cada argumento de um método covariante no programa Java gerado. Para cada parâmetro, se o tipo a introduzir na AST for diferente de *Object*, a informação que temos sobre o mesmo

será o seu nome completo. Eliminando-se os pontos das extremidades da sequência de caracteres que representa esse nome, resulta um nome de uma classe ou interface existente na hierarquia de classes do programa que está a ser tratado, pelo qual será substituído o tipo do parâmetro original no código JavaCO de entrada.

Assim, se o tipo em questão tiver o nome completo *".A.B."*, resultará para a substituição o tipo *"A.B"*. Note-se que o tipo aqui obtido será válido para o compilador de Java.

No caso de o tipo para usar na substituição representar uma classe definida dentro de um método, tem de se fazer algo mais. Como já foi referido no subcapítulo 6.5.2.3., esta classe inclui no seu nome completo o nome do método, precedido de um símbolo cardinal. A substituição usará, de entre os elementos que compõem o nome completo, todos aqueles que se encontrem após o nome do **último** método, exclusive.

Por exemplo, se o nome completo do tipo for *".A.#walk.B.#sit.C.D."*, o tipo a usar será *"C.D"*.

6.5.8. Estruturas usadas na implementação

Classe Globals

A classe *Globals* contém as duas *Hashtables* mais importantes do tradutor, denominadas de *classesAndInterfaces* e *methods* e que são usadas em todo o processo de tradução, armazenando toda a informação relevante sobre classes/interfaces e métodos, respectivamente.

Esta classe, para além de servir de base de dados para grande parte da informação recolhida aquando da iteração dos dois primeiros *walkers* pela AST, contém diversos métodos estáticos necessários em várias etapas da tradução.

Classe XClassOrInterface

Esta classe permite a criação de objectos que visam guardar a informação relevante, para os nossos objectivos, das classes e interfaces existentes no ficheiro JavaCO de entrada.

Os elementos de informação mais importantes que são guardados nos objectos da classe *XClassOrInterface* são: referência para o nó original da classe/interface na AST do programa; o nome completo da classe/interface; um valor booleano que indica se se trata de uma classe ou uma interface; os seus parâmetros de tipo, caso existam; a lista de todos os métodos definidos dentro da classe; a lista de todos os métodos herdados pela classe; a lista dos ascendentes directos na hierarquia de classes; a lista dos métodos que a classe/interface fornece aos seus descendentes através da herança. Toda esta informação vai sendo preenchida e actualizada nas fases 1 e 2 do processo de tradução, antes da terceira e última fase que corresponde à geração do código Java.

Métodos estáticos mais importantes

- *checkForCovariantErrorsOnEachClass( )*
  Este método é executado na fase 2 do processo de tradução, numa altura em que toda a informação relevante para as verificações de covariância se encontra completa. Dentro do conjunto de métodos definidos e herdados em cada classe, averigua-se a unicidade do nome de cada método covariante. Numa classe, se para um dado método covariante for descoberto na mesma classe um outro método definido ou herdado com o mesmo nome, considera-se um erro e adiciona-se uma mensagem à classe *Error*.

- *isSubtypeOf (String subtype, String supertype)*
  Este é um método booleano que testa a compatibilidade entre dois tipos. Retorna **true** caso se confirme que *subtype* é descendente de *supertype* na hierarquia de classes.

- *fillInherited( )*

  Este método é chamado após a iteração do segundo *walker* pela AST. Visa o preenchimento dos conjuntos de métodos herdados por casa classe, objectivo que será alcançado através da invocação do método *buildInherited* para todas as *XClassOrInterface* presentes na Hashtable de classes e interfaces.

Métodos de instância mais importantes

- *buildInherited( )*

  Como foi dito, esta classe possui uma variável para armazenar os métodos herdados pela classe/interface que representa. A informação sobre estes métodos, tal como a relativa aos métodos definidos por uma classe/interface, é necessária para as posteriores validações de covariância.

  O método *buildInherited* irá, para a classe/interface em questão, percorrer todos os seus pais na hierarquia. A cada um, irá pedir que forneça todos os métodos herdados e definidos (excepto os privados) através do método *supply*. Com as verificações devidas, estes métodos são agregados e irão representar os métodos herdados da classe/interface.

- *supply( )*

  O método *supply* não recebe argumentos e é invocado sempre que se está a construir o conjunto de métodos herdados de uma classe/interface, como descrito previamente. Este método irá retornar o conjunto de métodos a fornecer às classes que se encontrem no nível abaixo na hierarquia. Ao conjunto de métodos a retornar pelo *supply*, são primeiramente adicionados os métodos definidos na classe corrente (à excepção dos métodos privados). De seguida, serão adicionados os métodos herdados, provenientes de todos os pais, que por sua vez se vão obter através do método *buildInherited*.

Os dois últimos métodos possuem uma grande cumplicidade, visto que se chamam recursivamente entre si de modo a construir correctamente o conjunto de métodos herdados de cada classe. É de salientar que esta informação é essencial para algumas verificações a efectuar e que têm o intuito de fazer a validação da covariância do programa.

Note-se que diversas classes distintas podem solicitar o *supply* a uma mesma classe, no caso em que todas são suas subclasses. O conjunto de métodos a retornar pelo método *supply* será sempre o mesmo. Assim, na primeira vez que é criado este conjunto de métodos para retorno, é guardado como variável interna à classe *XClassOrInterface*, para que possa ser retornado directamente aquando de uma futura chamada do método *supply*.

Classe XMethod

Esta classe possibilita a criação de objectos que irão conter alguns dados sobre os métodos existentes no programa JavaCO de entrada. Obviamente que se trata de uma das mais importantes classes deste tradutor, facilitando o armazenamento da informação recolhida relativamente aos métodos, aspecto fundamental relativamente aos objectivos deste trabalho.

Os dados armazenados em cada objecto deste tipo são: a referência para o nó do método na AST; a referência para a *XClassOrInterface* onde se encontra o método; a informação sobre se o método é covariante; a informação sobre se o método é privado; se ocorrem parâmetros de tipo na sua assinatura; uma referência para o seu *MethodHeader*, classe que possui informação sobre a assinatura de um determinado método.

Método mais importante

- *belongs (Hashtable<Node,XMethod> met, boolean testCompatibleHeader)*
  O método *belongs* recebe como argumentos um conjunto de métodos, vazio ou não, e uma informação que nos indica se estamos a testar a compatibilidade ou igualdade do objecto *XMethod* corrente com os métodos presentes no conjunto referido.

Classe MethodHeader

A classe *MethodHeader* permite a criação de objectos que representarão as assinaturas de métodos. Estes objectos encontrar-se-ão associados aos métodos, sendo que um método apenas pode ter um *MethodHeader* associado. Apresentam-se como objectos de grande utilidade, usados para comparação entre métodos, quer seja igualdade ou compatibilidade.

Os principais dados que se podem encontrar num destes objectos são o identificador do método, a referência para o respectivo *XMethod* e duas listas, uma com os nomes dos tipos dos argumentos e outra com as referências para os nós desses argumentos na AST.

Métodos mais importantes

- *testIsSameOrCompatibleHeader(MethodHeader mh, boolean testCompatibility)*

  É um dos métodos mais importantes para a obtenção de resultados correctos na tradução final e trata da verificação de igualdade ou compatibilidade entre as assinaturas de dois métodos. *testIsSameOrCompatibleHeader* recebe como parâmetros um objecto do mesmo tipo, *MethodHeader*, e a indicação do tipo de teste a efectuar, igualdade ou compatibilidade.

  Quando se está a construir o conjunto de métodos herdados por uma classe/interface, através do método *buildInherited*, é criada uma estrutura de dados auxiliar à qual vamos adicionando os métodos pretendidos. Ao iterar sobre os métodos de cada pai da classe/interface, para adicionar um desses métodos ao conjunto final, testa-se a igualdade acima referida para saber se esse método já existe no conjunto. Testa-se a compatibilidade para averiguar se esse método é compatível com algum dos métodos definidos na própria classe/interface.

  Este método, *testIsSameOrCompatibleHeader*, começa por comparar os identificadores das duas assinaturas submetidas ao teste, e posteriormente os seus números de argumentos. Caso haja desigualdade num deles, as assinaturas não são

obviamente iguais ou compatíveis. Caso contrário, os comportamentos seguintes dependerão do tipo de teste que estamos a efectuar.

Se for a igualdade, será comparada a igualdade estrita entre cada par de tipos de argumentos com o mesmo índice nas duas assinaturas. Para além disso, estas só serão consideradas idênticas se ambas tiverem ou não tiverem o modificador **covariant**.

Se o teste for de compatibilidade, iterar-se-á também sobre os argumentos, averiguando-se se o tipo de cada parâmetro da assinatura do método corrente é subtipo do parâmetro correspondente na assinatura do outro método, caso se tratem de parâmetros tipo referência. Caso sejam tipos primitivos, só serão considerados compatíveis se forem rigorosamente iguais. Caso todos os parâmetros da assinatura corrente sejam compatíveis com os da segunda assinatura, importa certificarmo-nos que ambos os métodos são covariantes. Se um deles não for, iremos adicionar um erro à classe *Error* - este erro irá notificar posteriormente o utilizador que se esqueceu de colocar o modificador **covariant** na definição do método na superclasse, ou na redefinição do mesmo na subclasse. Sendo ambos covariantes, resta actualizar os objectos *BestTypes* de ambas as assinaturas, através da invocação do método *handleBestTypes*.

- *handleBestTypes (MethodHeader mh1, MethodHeader mh2)*
  Este método é invocado pelo método anteriormente descrito quando as duas assinaturas em causa são compatíveis. Como parâmetros são passados os objectos *MethodHeader* que representam essas assinaturas.
  A sua funcionalidade é a manipulação dos objectos *BestTypes* relativos a essas assinaturas, como será explicado de seguida.

Classe BestTypes

Esta classe define os objectos nos quais iremos guardar a informação necessária para a escolha dos melhores tipos a usar na substituição dos tipos dos parâmetros covariantes, na altura da geração do código Java de saída do tradutor.

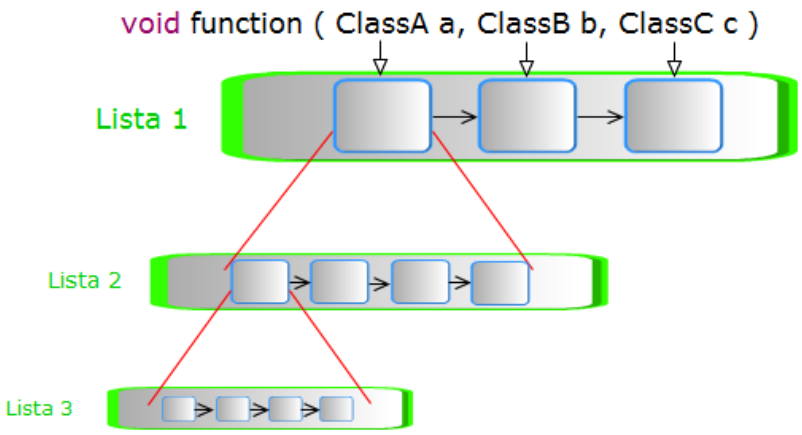
Estes objectos são criados quando, no primeiro *walker*, se detectam métodos covariantes. A referência para cada objecto será adicionada ao *MethodHeader* do método correspondente.

Numa primeira fase, a solução para a escolha dos melhores tipos para substituição foi a alternativa número 2, descrita no subcapítulo 6.5.6. A parte mais complicada da implementação desta alternativa reside na identificação, para cada argumento de cada método covariante com as suas redefinições, da colecção de tipos de onde se vai seleccionar a classe que é ascendente de todos estes. A descoberta do ascendente é um processo gradual que avança mais um pouco sempre que os métodos covariantes de dois dos tipos envolvidos passam o teste do método *testIsSameOrCompatibleHeader*.

Resumidamente, quando se verifica a existência de compatibilidade entre todos os tipos dos parâmetros de duas assinaturas de métodos covariantes, escolhe-se qualquer um dos dois objectos *BestTypes* que, após actualização dos tipos, passa a ser partilhado por ambas as assinaturas. Cada um destes objectos tem uma lista com os melhores tipos para substituição, cujo tamanho é fixo (desde a criação do objecto) e dado pelo número de parâmetros tipo referência – para parâmetros de outros tipos não há verificações a fazer relativamente à escolha de melhores tipos – das assinaturas a que se encontra associado. Os tipos presentes nesta lista vão sendo actualizados (escolhe-se sempre o que seja ascendente do outro) à medida que é solicitado o tratamento de dois objectos *BestTypes*, resultante da verificação de compatibilidade referida.

A alternativa adoptada foi a alternativa 3 descrita no subcapítulo 6.5.6., e para isso foi necessário trocar a lista anteriormente usada por uma lista de listas, que terá o mesmo tamanho da anterior. A parte mais complicada da implementação desta alternativa reside na identificação, para cada argumento de cada método covariante com as suas redefinições, de qual a lista das listas a intersectar. A construção da lista de listas é um processo gradual que avança mais um pouco sempre que os métodos covariantes de dois dos tipos envolvidos passam o teste do método *testIsSameOrCompatibleHeader*. Evidentemente que um método pode ter múltiplos argumentos, e portanto é natural que surja a necessidade de usar listas de listas de listas.

Na figura abaixo encontra-se descrita a estrutura de uma lista de listas de listas, aqui denominada por ***Lista 1***, de um objecto *BestTypes*.



Estrutura de um objecto *BestTypes*

A ***Lista 1*** possui um elemento para cada parâmetro do método. Cada um desses elementos é uma ***Lista 2***, onde vão sendo adicionadas as listas de ascendentes do parâmetro em questão; cada ocorrência diferente desse parâmetro nos métodos redefinidos contribui com uma lista de tipo ***Lista 3*** para a lista de tipo ***Lista 2***. Numa lista de tipo ***Lista 3***, estão guardados todos os ascendentes do tipo de uma ocorrência de um argumento dum método covariante ou sua redefinição.

O tamanho da ***Lista 2*** será variável. Começa por ter um elemento, contendo os ascendentes do parâmetro em questão do único método ao qual o objecto *BestTypes* se encontra associado na altura da sua criação.

Considere-se o método *function* da figura atrás, e duas hipotéticas redefinições (que não aparecem na figura), nas quais o primeiro parâmetro tem o tipo *ClassAA* e *ClassAAA*, respectivamente. Assuma-se a compatibilidade entre estes tipos e *ClassA*. O tratamento dos *BestTypes* entre a definição do método e a primeira redefinição, resultará na associação de um destes objectos a ambos os métodos – o outro não será eliminado, mas não mais será tido

em conta. A ***Lista 2*** relativa ao primeiro argumento e referida anteriormente, irá agora ter dois elementos – um contendo os ascendentes da *ClassA* e outro com os ascendentes da *ClassAA*. Quando se proceder ao tratamento entre este objecto e o *BestTypes* relativo à segunda redefinição, resultará um objecto cuja mesma ***Lista 2*** terá agora três elementos – será adicionado um elemento contendo os ascendentes de *ClassAAA*.

Recapitulando, quando verificada com sucesso a compatibilidade entre dois métodos, é chegada a altura de fazer o tratamento destes objectos. O que acontece é, basicamente, a união dos *BestTypes* correspondentes. Escolhe-se um destes objectos, da forma revelada anteriormente, que será aquele que no final estará associado às assinaturas de ambos os métodos e ao qual se adicionará a informação proveniente do outro objecto (cuja existência será futuramente ignorada).

Este tratamento de dois objectos *BestTypes* consiste então em juntar os ascendentes de cada um dos argumentos dos dois métodos envolvidos, da forma descrita relativamente ao primeiro parâmetro do método da figura e suas hipotéticas redefinições.

Note-se que todo o processo de actualizações destes objectos decorre antes da terceira e última fase da tradução. Nesta última fase, quando é necessária a substituição dos parâmetros covariantes, já todos os métodos covariantes e respectivas redefinições se encontram associados ao mesmo objecto *BestTypes*. Aí, para cada parâmetro, será feita a intersecção de todos os elementos da ***Lista 2*** que lhe está associado, de onde resultará uma lista com todos os ascendentes **comuns** aos tipos dos parâmetros cujos métodos estão associados a este objecto *BestTypes*. Destes ascendentes comuns escolher-se-á, caso exista, aquele que seja subtipo de todos os outros, e que portanto melhor se adequará à substituição.

Classe Parent

Aquando da recolha de dados sobre as classes e interfaces existentes no programa JavaCO de entrada, é indispensável associar a cada classe/interface a informação existente sobre os seus pais, quer sejam a classe estendida ou interfaces implementadas por uma classe, quer sejam as interfaces estendidas por uma interface.

Consideremos a declaração da classe *A*:

```
class A<X> extends B<X>
```

Importa guardar toda a informação que temos acerca do tipo *B<X>*. Como essa informação é mais do que o nome da classe em questão, foi criada esta classe, *Parent*, que nos vai dar a possibilidade de albergar todos os dados relevantes. Existe uma variável do tipo *String* para o nome completo da classe *B*, uma referência para a respectiva *XClassOrInterface* e uma lista para guardar os nomes dos seus parâmetros de tipo, caso existam. Neste exemplo, *X* representa a variável de tipo *X* da classe *A*.

Classe Error

Como já foi referido, não é objectivo deste tradutor detectar eventuais erros de Java no código recebido à entrada. Procuraram-se as melhores formas de contornar esse tipo de incorrecções para que, mais tarde, o compilador de Java possa fazer o seu trabalho e alertar o utilizador acerca dos erros de Java existentes.

Os erros que o tradutor irá reportar ao programador são aqueles que foram encontrados durante a recolha de informação e durante as validações, estando relacionados com a covariância (incluindo erros relacionados com o tipo **This**) e com as limitações do tradutor. Alguns exemplos são o esquecimento de um modificador **covariant**, tal como o seu uso em local indevido, a ocorrência de um método covariante e *overloaded* ao mesmo tempo, o uso da palavra **This** em local não adequado ou a utilização das palavras-chave **package** e **import**.

Caso um qualquer erro seja detectado, não faz sentido efectuar a transformação da árvore. Consequentemente, o tradutor não irá gerar o código Java respectivo. Havendo a possibilidade de diversos erros destes ocorrerem num mesmo programa, decidiu-se criar uma classe *Error*, partilhada por todo o fio de execução do nosso tradutor, tal como a classe *Globals*. Desta forma, cada vez que um destes erros seja detectado, irá ser adicionado à lista

estática de objectos do tipo *Error* existente dentro da própria classe. Desta forma o programador terá a informação precisa dos erros encontrados.

Após todas as verificações, proceder-se-á à transformação da AST, gerando-se o código Java de saída, caso a lista de erros esteja vazia. Caso contrário, serão mostradas ao utilizador as incorrecções detectadas.

6.5.9. Propriedades a destacar e limitações do tradutor

O tradutor implementado recebe à entrada um ficheiro com o formato JavaCO, produzindo um ficheiro com o mesmo nome e extensão “java”.

Esta versão do tradutor ainda não aceita múltiplos ficheiros de entrada.

Não há suporte para classes de biblioteca JavaCO. Neste momento, a utilidade das palavras ***package*** e ***import*** restringe-se à utilização de classes de biblioteca do Java.

7. This (apenas nos parâmetros)

Como foi revelado anteriormente, o tipo **This** encontra-se intimamente ligado ao uso de covariância na linguagem JavaCO. Tendo um significado flexível ao longo da hierarquia de classes, **This** representa o tipo da instância da classe onde ocorre e terá de ser reinterpretado nas subclasses, sempre que aparece em código herdado. Já dentro de cada objecto, o significado de **This** é fixo e representa o tipo do próprio objecto.

Retomando o exemplo dos animais, reescreve-se de seguida o método binário reprodução, de uma forma mais geral, através do uso deste novo tipo. Usando esta nova forma, conseguimos alcançar a expressividade e significado desejados.

JavaCO

```
class Animal {
  covariant void reprodução(This animal){
  }
}
```

Repare-se que, com o uso do tipo **This**, cujo significado se irá alterando consoante a subclasse que herde o método, conseguimos obrigar o programa a ter um comportamento aproximado à realidade e de acordo com os nossos propósitos, onde um determinado animal só se pode reproduzir única e exclusivamente com animais da mesma espécie.

A palavra reservada **This** pode ocorrer, nesta primeira fase, apenas na assinatura de métodos covariantes e como tipo de (um ou mais) argumentos.

Interessa salientar que quando se redefine um método com argumento do tipo **This**, não se torna obrigatório a repetição do tipo **This** na subclasse. A validação dos tipos é feita, permitido que ao longo da hierarquia o tipo vá alternando entre **This** e tipos concretos, desde que a relação de subtipo seja sempre respeitada. Para efeito das validações, troca-se sempre o tipo **This** pelo nome da classe envolvente.

Um pequeno exemplo. No código que se segue, no ponto assinalado com **X**, dentro da classe *B*, podem ser usados os tipos *A*, *B*, *C* e **This**, que fica garantida a consistência da colecção de quatro classes. Contudo, não pode ser usado o tipo *D* porque a redefinição do método *f* na classe *C* ficaria inválida.

JavaCO

```
class A {
    covariant void f ( This a ) { ... }
}

class B extends A {
    covariant void f ( X a ) { ... }
}

class C extends B {
    covariant void f ( This a ) { ... }
}

class D extends C {

}
```

O modificador **covariant**, citado no capítulo anterior, irá também autorizar o uso do tipo **This**. Onde quer que apareça, **covariant** assinala sempre a possibilidade de covariância dos argumentos.

7.1. MyType

A maioria das linguagens de programação orientadas a objectos não inclui nenhuma palavra-chave como *MyType* ou This para referir o tipo de *this*. Mas há algumas excepções.

Trellis/Owl e PolyTOIL são duas linguagens, com tipificação estática, que incluem a palavra-chave *MyType* [K. B. Bruce et al. 03].

Trellis/Owl [Schaffert et al. 86] foi uma das primeiras a incluir esta construção. Devido à possibilidade de aparecimento de falhas no sistema de tipos, esta linguagem exige que todas as subclasses também gerem subtipos. Esta necessidade restringe o uso de *MyType* aos tipos de retorno dos métodos, sendo que a sua ocorrência como tipo de parâmetros de métodos não iria resultar em subtipos.

Na linguagem PolyTOIL, o suporte para *MyType* também tem consequências no seu sistema de tipos. Nesta linguagem a ocorrência de MyType nos argumentos dos métodos implica a perda da relação de subtipo, e logo, o uso de polimorfismo nas classes em que MyType é usado nos argumentos.

Como foi dito antes, a linguagem Eiffel suporta uma construção com o mesmo intuito, rejeitando no entanto as restrições impostas em Trellis/Owl e as desvantagens em PolyTOIL, permitindo o uso de *like Current* como tipo de parâmetros de métodos sem quebrar a relação de subtipo.

A introdução da palavra-chave **This** em JavaCO é feita tirando partido de verificações de tipo efectuadas dinamicamente. Em JavaCO, o uso de **This** no tipo dos argumentos permite que as subclasses continuem a gerar subtipos.

7.2. Métodos binários

Um método binário é um método que contém um ou mais parâmetros do mesmo tipo do objecto que recebe a mensagem.

A implementação de métodos binários em linguagens tradicionais orientadas a objectos normalmente traz problemas no que diz respeito a relações entre tipos e classes no contexto de herança e a necessidade de acesso privilegiado à representação interna dos objectos.

Relativamente ao primeiro problema, o tipo **This** resolve essa questão. Um bom exemplo disso é aquele apresentado no início do capítulo anterior, em que o método *reprodução* tem de ir sendo explicitamente redefinido ao longo da hierarquia das subclasses de *Animal*. Com a introdução do tipo **This**, essa necessidade desaparece pois ao longo da hierarquia este tipo será reinterpretado, adaptando-se assim a cada subclasse.

Relativamente ao segundo problema, a linguagem JavaCO não oferece nada de novo. Para aceder à parte privada dum objecto de tipo **This**, o programador terá de definir e usar selectores e modificadores.

7.3. Transformações associadas ao tipo This

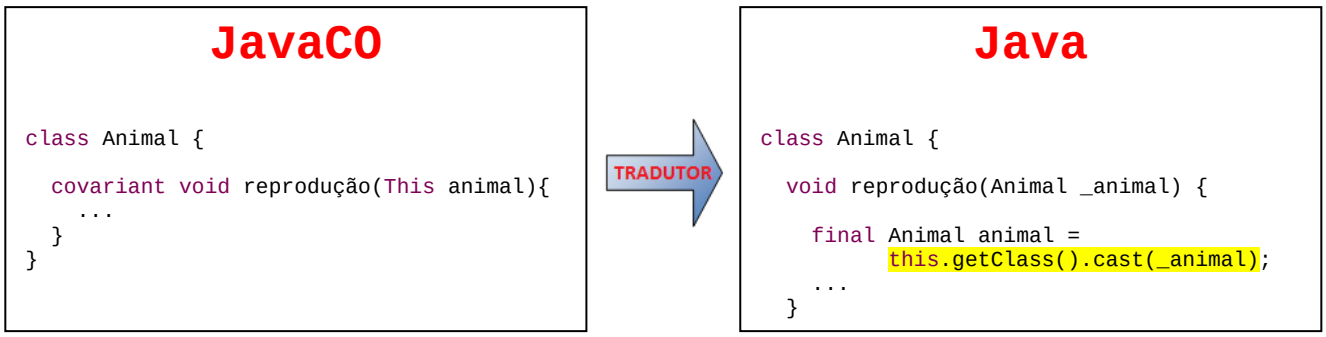
A transformação que ocorre num método quando o tipo do parâmetro covariante em questão é **This**, traz algumas alterações em relação ao que acontece aquando da ocorrência de um parâmetro de tipo diferente.

Como mencionado previamente, o principal objectivo da introdução do tipo **This** é a sua flexibilidade e reinterpretação ao longo da hierarquia de classes de um programa, à medida que os métodos onde ocorre são herdados em subclasses.

Desta forma, a validação do argumento recebido não é feita através do operador habitual de *cast*, mas sim o método reflectivo *cast( )*, disponível na classe *Class*. Note-se também que *this.getClass()* irá sempre retornar o tipo da classe onde se está a definir esta variável, mudando o seu significado consoante a subclasse para onde o método seja herdado.

Outro pormenor a ter em conta, é a obrigatoriedade de declarar a variável como sendo **final**. Tal como acontece no caso normal de covariância, a variável criada na tradução vai ter o mesmo nome do parâmetro original do método, no *input* em JavaCO. Ora se estamos a garantir que, neste caso, essa variável tem o mesmo tipo da classe em que o método está a ser herdado, seria inválido permitir que essa variável fosse alterada no corpo original do método, deixando em aberto a possibilidade de esta ser afectada com um objecto de qualquer outro tipo.

Encontra-se exemplificado de seguida o *output* da tradução de um método que contenha o tipo covariante **This** no conjunto dos seus parâmetros.



Nesta primeira fase, o tipo **This** poderá aparecer apenas nos argumentos de métodos covariantes. Portanto, a palavra **This** é proibida em todos os outros contextos. Concretamente: como identificador de qualquer entidade, nomeadamente, classe, interface, métodos, etc; como argumento de instanciação de uma classe genérica; como limite superior de um **extends**, etc; como tipo do resultado de métodos; como tipo de qualquer variável; etc.

7.4. Armazenamento de informação sobre parâmetros com tipo This

O aparecimento do tipo **This** nos parâmetros de métodos terá obviamente algumas implicações que teremos de ter em conta ao longo de toda a implementação. Quando se está a recolher informação da AST, com vista a armazenar todo o conteúdo indispensável para verificações futuras, não pretendemos descartar de imediato a ocorrência do tipo **This**, trocando desde logo o seu significado com o nome da classe/interface em que surgiu.

Quando pretendemos guardar o nome completo do tipo de um parâmetro de um método e nos depararmos com o tipo **This**, o elemento guardado será constituído pelo nome completo da classe/interface onde ocorre este parâmetro, precedido por *".This."*. Desta forma, quando em fases posteriores se proceder à análise deste tipo, sabemos que estamos perante um parâmetro do tipo **This** e temos como informação adicional o nome completo da classe ou da interface em que foi detectado.

7.5. Validações relativas ao tipo This

Como adição às verificações que, durante a primeira fase do processo de tradução, o primeiro *walker* já vinha fazendo, como por exemplo a validação dos modificadores **covariant** existentes no programa, este irá agora validar também todas as ocorrências do tipo **This**. Caso seja detectado o aparecimento desta palavra num contexto proibido, será gerado o respectivo erro.

8. This (também no resultado e variáveis)

Com vista a aproximar o JavaCO das funcionalidades da linguagem Eiffel, decidiu-se liberalizar o uso do tipo **This**. Significa isto que agora o programador poderá também fazer uso deste novo tipo tanto na declaração de variáveis locais e de instância como no retorno dos métodos.

O uso do tipo **This** nas variáveis e tipo de retorno tem um potencial de uso abaixo do que poderá parecer à primeira vista. Isto porque o tipo **This** é genérico e todas as expressões de tipo **This** que se escrevam têm de produzir instâncias das classes para onde são herdadas, quaisquer que sejam essas classes. Considere a seguinte definição de variável local:

```
This var = exp;
```

Nesta definição há poucas variantes de concretização de *exp*. Uma possibilidade é a própria expressão **this**. *exp* também pode corresponder à invocação de um método que retorne **This**. Não existem muito mais alternativas. Nomeadamente, não se pode atribuir à variável expressões de qualquer tipo diferente de **This**, nem mesmo do tipo da classe envolvente.

Portanto, o programador tem de aprender a usar o tipo **This** e habituar-se a certas restrições, que numa primeira fase podem parecer pouco intuitivas.

8.1. Validação dos novos usos de This

A segurança associada a estes novos usos da palavra **This** poderia ser efectuada inserindo testes de tipo dinâmicos no código traduzido. No entanto, nesta parte do trabalho foi possível fazer melhor. Na realidade, foi possível validar este aspecto estaticamente, não directamente no tradutor, mas sim indirectamente através da geração de código Java criado com o objectivo único de ser validado pelo compilador de Java. Portanto, qualquer problema que haja com os novos usos de **This** não dá origem a um erro detectado pelo tradutor, mas dará sim, mais tarde, um erro detectado pelo compilador de Java.

O esquema de validação que foi criado baseia-se directamente na ideia de que **This** representa o tipo da classe corrente ou de qualquer subclasse. Desta forma, para validar o **This** dentro de uma classe *A*, limitamo-nos a gerar um duplicado dessa classe com o nome modificado, *A_Validate*, e definimos essa classe como subclasse de *A*. Depois, a palavra **This** é trocada em cada uma das classes pelo respectivo nome. Desta forma, consegue-se que o código de cada método onde **This** ocorre seja validado na classe *A* e na classe *A_Validate*. Repare que *A_Validate* representa uma subclasse de *A qualquer*. Por outras palavras, não é preciso fazer a validação para as infinitas potenciais subclasses de *A*; basta fazer essa validação para uma subclasse genérica *A_Validate*.

Desta forma, optou-se pela criação de classes auxiliares para a validação do tipo **This** no caso das variáveis e no retorno dos métodos. Antes do algoritmo central do tradutor exercer as suas funções, serão adicionadas classes de validação à AST do programa de entrada.

Note-se que, com o uso destas classes de validação, desaparece a necessidade de impor que os argumentos de tipo **This** sejam *read-only* e portanto a variável auxiliar que é usada na implementação deste caso deixa de usar o modificador **final**, ao contrário do que era necessário no uso de **This** apenas nos argumentos dos métodos covariantes.

8.2. Descrição detalhada do método adoptado

Para permitir a liberalização do uso do tipo **This** como pretendido decidiu-se então conceber uma classe de validação para cada classe onde ocorre este tipo (nas interfaces não é necessário efectuar o mesmo tratamento visto que ai não existem variáveis definidas, bastando trocar **This** pelo seu significado no retorno dos métodos). Todo o interior da classe será mantido, sendo que se adicionará "_Validate" ao seu nome, mantendo-se quaisquer parâmetros de tipo existentes. A nova classe herdará da original, onde também irão ser mantidos os parâmetros de tipo originais.

Desta forma, uma classe *Lista<Integer>* definida no programa de entrada causa o aparecimento de uma nova classe *Lista_Validate<Integer> extends Lista<Integer>*. Esta classe irá ser gerada antes de todo o tratamento efectuado pelos *walkers* principais e será colocada ao lado de todas as restantes classes e interfaces (na AST), antes do tratamento referido.

8.3. Implicações da palavra-chave final

O processo descrito para a validação da declaração de variáveis de tipo **This**, bem como retorno deste tipo por parte de um método, é simples e útil na teoria, mas há cuidados a ter na duplicação de classes.

Como sabemos, um método que seja declarado em Java com o uso da palavra-chave **final** não pode ser *overriden* por uma subclasse. Assim sendo, é necessário que as classes a duplicar para validação não tenham qualquer método nestas circunstâncias. Para isso, antes de gerarmos as classes de validação, iremos duplicar qualquer classe que possua no seu corpo o modificador **final** (e obviamente o tipo **This** em variáveis ou no retorno de um método seu). Esta classe será em tudo igual à original, excepto no seu nome, ao qual se acrescentará os caracteres "_noFinal", e obviamente na ausência de modificadores **final**.

Note-se que para efeito da validação não levanta qualquer problema a omissão destes modificadores. A classe original manter-se-á igual, pelo que não estamos a eliminar a restrição imposta pelo programador através do uso de tal modificador. Ao invés disso, estamos apenas a gerar uma classe que, na ausência de modificadores **final**, pode ser duplicada para validação dos tipos **This** existentes fora das assinaturas de métodos.

8.4. Implementação

De entre as formas possíveis para geração das novas classes pretendidas, a que mostrava mais simplicidade era a geração de uma nova AST do programa. Modificando-se os nós da própria árvore, temos a possibilidade de alterar as classes existentes efectuando as pequenas modificações necessárias, obtendo-se desta forma as classes pretendidas.

Uma hipótese seria criar internamente uma nova classe, com o nome e a superclasse pretendidos, copiando todo o restante conteúdo, entendam-se os nós interiores à classe. No entanto, não se optou por este caminho porque, por um lado, colocar nesta nova classe referências para todos os nós encontrados no interior da classe original seria de todo desaconselhável, devido aos danos colaterais que poderiam advir de uma alteração posterior em qualquer desses nós. Por outro lado, tentar fazer cópias de todos esses nós pode considerar-se totalmente impraticável à partida, porque seria necessário implementar a funcionalidade de copiar cada um dos mais de 800 tipos de nós que podem ocorrer numa AST gerada pelo SableCC para a gramática JavaCO.

Como não se encontrou forma simples de duplicar as sub-árvores da AST que representam classes, para obter essas novas sub-árvores usa-se um método muito básico: aplica-se o *parser* novamente ao ficheiro de entrada para gerar um exemplar novo da AST, que vai servir como fonte de novos nós para serem alterados e adicionados à AST principal. Chamamos AST principal à árvore sobre a qual actuarão os quatro *walkers* já explicados nos capítulos anteriores.

A criação duma AST duplicada torna a implementação bastante ineficiente. Pior ainda: existem ainda situações que podem obrigar a criar até mais duas ASTs duplicadas, podendo ser atingido, na pior das hipóteses, o máximo de três ASTs, para além da AST principal.

8.4.1. Walker gerador de classes noFinal

Este *walker* irá percorrer uma AST, alterando o nome de cada classe que contenha o tipo **This** e o modificador **final** no seu interior. Após iterar sobre todas as classes, guardará numa lista os nós das classes alteradas, e consequentemente toda a informação a elas associada.

Esta lista estará disponível para posterior adição a outra AST.

Veremos mais adiante o contexto de utilização deste *walker*.

8.4.2. Walker gerador de classes Validate

Durante a iteração sobre uma AST, este *walker* irá alterar o nome de cada classe que contenha o tipo **This** e **não** contenha o modificador **final** no seu interior. No fim, armazenará numa lista os nós das classes alteradas e toda a informação nelas incorporada.

Esta lista estará disponível para adição à AST principal.

8.4.3. Funcionamento geral

1) Inicialmente é criado o primeiro exemplar da AST, sobre o qual itera um *walker* gerador de classes *noFinal*. No fim, a lista de classes alteradas pode estar ou não vazia.

2) É criado o segundo exemplar da AST, que é percorrido por um *walker* gerador de classes *Validate*. No início da iteração são adicionados a esta AST os nós existentes na lista

produzida no *walker* anterior. No fim, a lista de classes alteradas por este *walker* pode estar ou não vazia.

3) Caso a lista produzida no primeiro *walker* não esteja vazia (existência de classes que contenham simultaneamente o tipo **This** e o modificador **final**), é criado o terceiro exemplar da AST sobre o qual itera um terceiro *walker*, gerador de classes *noFinal*. Este produz uma lista idêntica à que foi criada no primeiro *walker*, com os nós das classes alterados, cujos nomes terminam em "_noFinal".

4) É criado o quarto e último exemplar da AST, à qual são adicionados os nós contidos na lista do ponto *2)* e, caso ocorra o ponto *3)*, os nós dai resultantes. Esta última AST é aquela que passará a ser considerada como AST principal, e é sobre ela que serão efectuados os processamentos dos quatro *walkers* referidos nos capítulos anteriores.

8.4.4. Uso de dois walkers geradores de classes noFinal

Como já foi referido, durante a actuação destes *walkers* estão a ser alteradas as árvores sobre as quais iteram. Significa isto que quando um nó de uma classe é alterado, o nó contendo a classe original deixa de estar disponível.

Consideremos o caso em que temos uma classe *Carro*, que contém no interior pelo menos um modificador **final** e uma criação de variável do tipo **This**.

Após a execução do ponto *1)* acima referido sobre a AST deste programa, o primeiro *walker* vai conter na sua lista o nó com a classe *Carro_noFinal*. A AST sobre a qual actuará o *walker* do ponto *2)*, irá por sua vez conter a classe *Carro* original e a classe *Carro_noFinal*, que lhe é adicionada.

Sendo que a classe original contém pelo menos um modificador **final** e um tipo **This**, não será transformada no ponto *2)*. A classe *Carro_noFinal*, por sua vez, contém o tipo **This** e não contém qualquer modificador **final**, pelo que será a única a ser alterada nesta etapa. No

final da passagem deste segundo *walker*, a lista que este possui irá conter o nó da classe *Carro_noFinal_Validate*.

Esta classe vai ser então adicionada à nova e última AST criada, sobre a qual actuarão os *walkers* principais. Ora a classe *Carro_noFinal_Validate* herda da classe *Carro_noFinal*, cujo nó foi entretanto alterado para a criação da classe *Carro_noFinal_Validate*.

A necessidade de adicionar também a classe *Carro_noFinal* a esta última árvore, obriga-nos a proceder de acordo com o ponto *3)*. Só assim a AST a usar pelos principais *walkers* da tradução se encontrará completa e preparada para todas as validações a efectuar.

8.5. Considerações sobre o método adoptado

Como já foi referido, esta é uma implementação pouco eficiente. As causas são o facto de ter de se percorrer a AST do programa de entrada mais vezes do que aquelas necessárias para a tradução, e a possibilidade de o código Java final ter até o triplo da dimensão inicial.

Caso se verifique esta última possibilidade, ela conduzirá a uma compilação mais morosa do programa de saída (por parte do compilador de Java) do que se não tivéssemos usado este mecanismo. No entanto, em tempo de execução não causa mais lentidão visto que apenas existem algumas classes a mais (classes de validação), que nunca vão ser usadas.

Apesar das desvantagens mencionadas, decidimos implementar este mecanismo quando descobrimos que se podiam efectuar as validações necessárias recorrendo ao próprio Java.

9. Genéricos e Parâmetros de tipo

Em JavaCO, o tratamento covariante das classes genéricas e dos parâmetros de tipo é muito limitado, sendo isso uma consequência das próprias limitações do Java relativamente a esse género de entidades.

9.1. Tipos genéricos e suas limitações

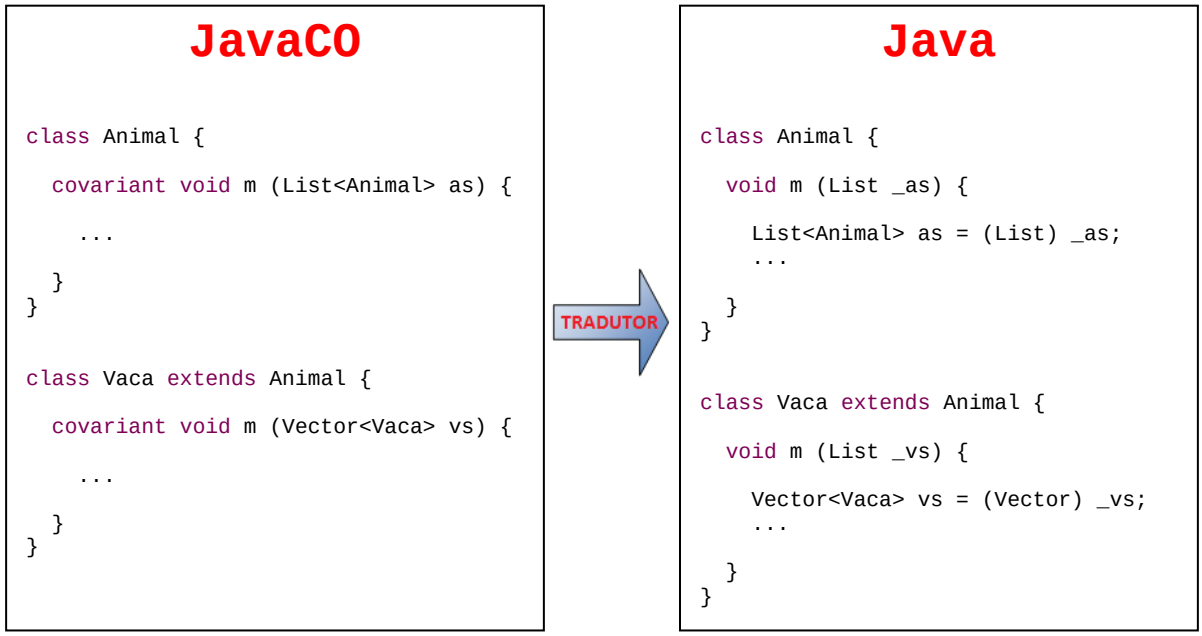
A linguagem JavaCO tenta imitar o Eiffel no que diz respeito aos tipos genéricos, ou seja, usa uma relação de subtipo entre classes genéricas que é baseada em covariância. Eis alguns exemplos de pares de tipos que pertencem à relação de subtipo:

Vector<Animal> **<:** *List<Animal>*

Vector<Vaca> **<:** *Vector<Animal>*

Vector<Vaca> **<:** *List<Animal>*

A linguagem JavaCO permite o uso de tipos destes nos argumentos de métodos covariantes, sendo esta relação de subtipo validada estaticamente pelo tradutor. No código gerado para cada método, e como tem sido feito até agora, é colocada uma instrução de validação dinâmica de tipo à entrada do corpo do método. Vejamos um exemplo de tradução:



Nesta tradução pode observar-se a omissão dos argumentos de alguns tipos genéricos do lado direito. No caso da omissão nos argumentos dos métodos, isso tem por objectivo o suporte de covariância, já que o Java não suporta covariância nos tipos genéricos. Na linha do que tem vindo a ser feito neste trabalho, a validação dos argumentos faz-se usando um *cast* na primeira instrução dos métodos. Poderá parecer estranho omitirem-se também os argumentos dos tipos genéricos dentro do *cast*, mas a verdade é que não serviria de nada incluir tais argumentos de tipo porque o Java os ignoraria. Este facto é uma consequência da forma como o Java trata os tipos genéricos, omitindo a informação de tipo dos parâmetros de tipo em tempo de execução.

Encontramos assim uma limitação da linguagem JavaCO que é a seguinte: apesar dos nossos métodos covariantes validarem com eficácia a parte “exterior” dos tipos genéricos, quanto aos parâmetros de tipo “interiores” não há qualquer possibilidade de validação dinâmica. Esta situação pode ser interpretada como a prova de que a linguagem JavaCO tem um sistema de tipos inseguro quando estão em causa métodos covariantes com argumentos de tipos genéricos. Mas temos de considerar que situações semelhantes ocorrem no Java “normal”, sendo por isso que esta linguagem está sempre preparada para detectar dinamicamente erros de tipo, gerando as correspondentes excepções.

Seguindo um dos princípios que foram evidenciados na secção 5.3, a parte exterior dos tipos genéricos é validada o mais cedo possível, logo à entrada dos métodos. No entanto, esse princípio é violado, pela única vez neste trabalho, na parte "interior" dos tipos genéricos. Existe a possibilidade de alguma violação do sistema de tipos não ser apanhada logo à entrada dum método, podendo a correspondente excepção ser gerada mais adiante, em qualquer parte do programa.

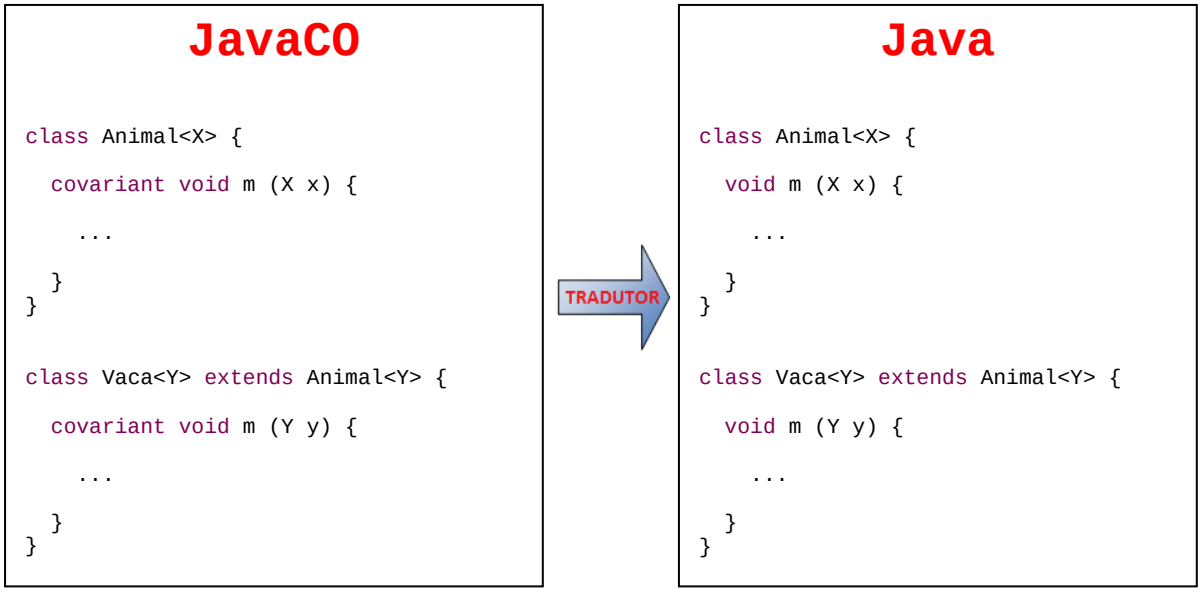
Então qual deve ser a atitude do programador perante esta situação? O programador deve continuar a assumir a regra da covariância para tipos genéricos na organização das suas ideias. No entanto, tem que ter um cuidado especial quando usa esses tipos porque sabe que algumas situações de violação do sistema de tipos são assinaladas de forma tardia.

Para terminar, convém referir que esta forma de tratamento dos tipos genéricos em JavaCO não é a única possível. A alternativa seria simplesmente não suportar o uso de covariância para argumentos de tipos genéricos, voltando à regra da invariância do Java.

9.2. Parâmetros de tipo e suas limitações

Relativamente aos parâmetros de tipo, a situação é mais radical do que a apresentada na secção anterior. Na linguagem Java não é permitido fazer um *cast* para um parâmetro de tipo, novamente porque a informação de tipo associada é apagada em tempo de execução. Portanto, para parâmetros de tipo, estamos impedidos de inserir nos métodos os já tradicionais *casts* que fazem a validação dos argumentos. Resulta daqui uma decisão drástica de não suportar argumentos covariantes para parâmetros de tipo. Quando o parâmetro de tipo é usado como argumento de um método covariante, na redefinição é obrigatório voltar a usar exactamente o mesmo tipo. Ou seja, em JavaCO quando estão em causa parâmetros de tipo regressa-se à regra da invariância do Java normal.

Esta situação tem por efeito simplificar um pouco a implementação, já que o tradutor não tem de se preocupar com os limites superiores (*upper bounds*) dos parâmetros de tipo.

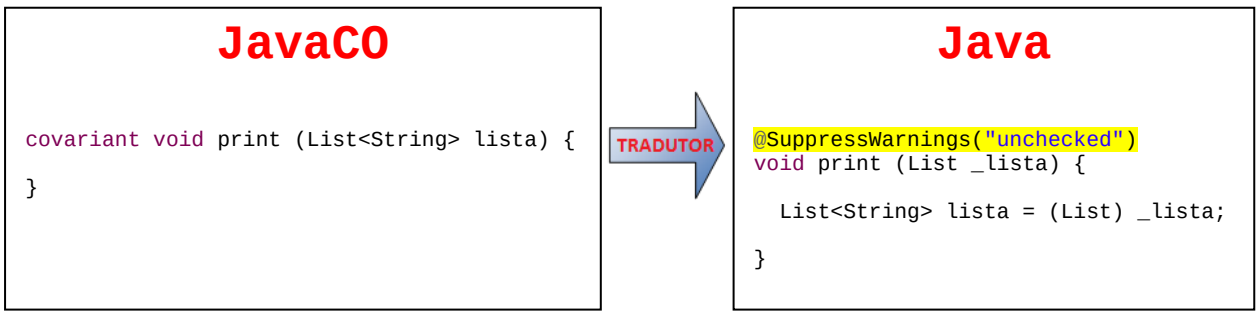


Como se pode ver no exemplo apresentado, não é efectuada qualquer alteração no código relativamente a parâmetros de tipo.

9.3. Problema com cast

Uma consequência do uso do *cast* acima mencionado é a geração de *warnings* do Java, neste caso relativo à detecção de uso de operações não verificadas ou inseguras.

Para não reportar ao utilizador este tipo de *warnings*, visto que é um aviso do nosso conhecimento e irrelevante em função dos objectivos do trabalho, estes serão suprimidos. Para tal, antes de qualquer método que contenha um ou mais argumentos cujos tipos tenham parâmetros genéricos, será inserida a anotação do Java cuja função é eliminar estes avisos.



9.4. Particularidade do supply atendendo aos parâmetros de tipo

Como descrito anteriormente, o método *supply* fornecido pela classe *XClassOrInterface* irá permitir a obtenção do conjunto de métodos que uma classe/interface fornecerá a todas as suas subclasses, sejam eles métodos não privados definidos na própria classe/interface, sejam métodos herdados dos pais.

Mas há que ter em atenção os casos onde os argumentos dos métodos covariantes são parâmetros de tipo. Foquemo-nos no seguinte exemplo, escrito em JavaCO:

JavaCO

```
public class A<X> {
  public covariant void m (X arg) { }
}
public class B<Y> extends A<Y> {
}
```

Note-se que ambas as classes declaradas têm um parâmetro de tipo. Sendo *m* um método público da classe *A*, será herdado pelas suas subclasses e consequentemente pela classe *B*.

Se a classe *B* herdasse o método referido com o argumento do tipo *X*, não iria reconhecer este tipo visto que o mesmo é um parâmetro interior à classe *A*. Tal como acontece no Java, também o tradutor terá a capacidade de instanciar os parâmetros da superclasse, substituindo-os adequadamente. Para isso, quando um objecto do tipo *XClassOrInterface* representa uma classe com parâmetros de tipo, o método *supply* que invoca não é o habitual, mas sim aquele que recebe como argumento a lista que contém os caminhos completos dos seus parâmetros de tipo.

No exemplo anterior, a *XClassOrInterface* que mantém a informação sobre a classe *B* contém na sua lista de nomes completos de parâmetros de tipo o elemento "*.B.Y.*", que representa o parâmetro *Y*. Na construção do conjunto de métodos herdados, será invocado o método

A.supply ([.B.Y.]), que retornará os métodos herdados e definidos (não privados) da classe *A*, onde o seu parâmetro *X (".A.X.")* será, onde quer que ocorra, substituído por *".B.Y."*.

9.5. Representação interna dos parâmetros de tipo

Em Java, um parâmetro de tipo é sempre tratado de uma forma distinta de uma classe. Por exemplo, é proibido escrever o nome completo *"A.X"* no caso de *X* ser o nome de um parâmetro de tipo da classe genérica *A*. A regra geral diz que o nome de um parâmetro de tipo nunca pode figurar como parte de um caminho completo.

No entanto, conseguimos simplificar bastante o processamento dos parâmetros de tipo, definindo para cada um deles uma classe interna vazia com o mesmo nome. Desta forma conseguimos reduzir o tratamento dos parâmetros de tipo ao tratamento das classes que já estava implementado.

Por exemplo, na classe *A<X>*, o parâmetro *X* é convertido numa classe interna vazia cujo nome completo é *".A.X."*, e esse será também o nome completo do parâmetro.

9.6. Particularidade na substituição de tipos

No momento imediatamente anterior à geração de código, os parâmetros de tipo são representados, como se fossem classes, pelos seus nomes completos, por exemplo *".A.X."*. O nome completo inclui um caminho que reflecte o imbricamento das classes que envolvem o parâmetro declarado. Neste caso, no código Java gerado, o tipo a usar será dado pelo nome localizado entre os últimos dois pontos do nome completo, que representa o nome básico do parâmetro de tipo.

Se a classe *A* tiver um parâmetro de tipo *X*, o nome completo do parâmetro será *".A.X."*. Na substituição usar-se-á o nome básico *X*.

9.7. Funcionalidades adicionadas

Para suportar tipos genéricos e parâmetros de tipo, foi necessário adicionar novas funcionalidades a diversas classes.

Vejamos primeiro o caso da classe *Globals*. Foram adicionados alguns métodos auxiliares cujo intuito é o tratamento de *Strings*. Alguns exemplos são: a separação dos parâmetros de tipo declarados por uma classe ou interface, para armazenamento dessa informação; a remoção dos limites superiores desses parâmetros, visto que não precisamos desses dados; a eliminação dos argumentos de tipo de um tipo, para introdução nos *casts* a utilizar na criação da variável auxiliar que ocorre na transformação de um método com argumentos covariantes, como já foi referido anteriormente.

No caso da classe *XClassOrInterface*, destacamos o aumento da funcionalidade do método *isSubtypeOf*. Este método passou a poder receber como argumentos tipos genéricos, e passou a definir regra de covariância para os argumentos dos tipos genéricos que foi apresentada no início deste capítulo.

10.Trabalho futuro

Algumas melhorias poderão ser feitas futuramente com vista à obtenção de um tradutor mais completo e que suporte mais funcionalidades. As mais importantes são descritas nesta secção.

10.1. Suporte de leitura de um conjunto de ficheiros em JavaCO

Como foi referido, o tradutor recebe à entrada um ficheiro com extensão ***.javaco*** que poderá conter diversas classes, interfaces e métodos.

Autorizando o uso da palavra-chave **package**, o utilizador iria ter a possibilidade de criar *packages* e assim organizar os ficheiros do seu programa em directorias tal como acontece na linguagem Java, através do uso desta palavra-chave. Esta é uma boa forma de organizar as classes programadas consoante as suas funcionalidades, usabilidade e categorias a que devem pertencer.

Sugestão de implementação

A implementação desta funcionalidade no tradutor produzido teria algumas implicações. A mais importante passa pela nomeação das classes e interfaces, cujos nomes completos passariam a integrar toda a informação que lhes está associada relativamente a *packages*.

Assim, uma classe *A* declarada dentro do *package world*, teria como nome completo *".world.A."*.

10.2. Possibilidade de herdar de classes JavaCO de biblioteca

Para expandir os horizontes da nossa nova linguagem, seria necessário permitir a criação de classes JavaCO de biblioteca bem como a possibilidade de herdar das mesmas. No entanto, para se dar ao utilizador essa possibilidade, há que ter em conta que estas classes de biblioteca, após traduzidas e compiladas em Java, terão de resultar em ficheiros ".class" com a informação dos métodos covariantes.

Sugestão de implementação

Uma forma de colocar esta informação nos programas JavaCO é através do mecanismo de metadata existente em Java, e que permitiria colocar anotações nos métodos pretendidos, nomeadamente aqueles declarados como sendo covariantes.

10.3. Melhorar a eficiência do compilador

Podendo haver diversos aspectos onde a eficiência da solução implementada não é a mais satisfatória, sem dúvida que o maior problema reside nas leituras adicionais do ficheiro de entrada, que ocorrem por três vezes. Isto deve-se ao facto de necessitamos, em alguns casos, de duplicar algumas classes do programa de entrada.

Sugestão de implementação

Com vista a resolver este problema, poder-se-ia tentar usar a serialização do Java para duplicar os nós e ramos da AST necessários. Desta forma o ficheiro de entrada seria lido uma só vez, e seriam retirados os três *tree walkers* extra que actuam no início do processo de tradução.

10.4. Indicação do número de linha no ficheiro original JavaCO de cada erro encontrado

Actualmente, caso sejam detectados erros pelo compilador de Java aquando da execução do programa traduzido, estes serão identificados a partir da linha do código Java onde ocorrem. Seria obviamente expectável a identificação dos erros de acordo com os números das linhas de código no ficheiro JavaCO original. Infelizmente, o Java não possui nenhuma directiva do género “#line” como ocorre nas linguagens C e C++, e que fornece números de linha para mensagens do compilador.

Assim sendo, o melhor que se pode fazer passa por modificar o código Java após a tradução, através da inserção/remoção de linhas de código em branco bem como da alteração da formatação do código, para que os números de linha passem a coincidir com os do código original. Alternativamente, pode ser alterada a forma de tradução, por forma a que os códigos original e final coincidam relativamente aos números de linha.

10.5. Adição de outros mecanismos da linguagem Eiffel

Sendo que este trabalho tem como fonte de inspiração os mecanismos da linguagem Eiffel, é de considerar a possibilidade de adicionar à linguagem JavaCO mais alguns mecanismos marcantes desta linguagem.

Um dos mais importantes seria a Programação por Contracto, visto que este mecanismo permite controlar o estado de todos os objectos que constituem o sistema e fornece ao programador um controlo sistemático da ocorrência de erros. Este mecanismo da linguagem Eiffel é mais poderoso que as asserções disponibilizadas pelo Java.

Poderia ser também interessante acrescentar à nossa nova linguagem o mecanismo de erradicação de chamadas sobre *null*, chamadas estas que são uma das principais fontes de instabilidade do software. Este mecanismo poderia também ser implementado por via de verificações dinâmicas de tipo.

11. Conclusões e análise crítica

Neste trabalho foi implementada a linguagem JavaCO, uma variante da linguagem Java cujo sistema de tipos é baseado em covariância. À imagem do que acontece em Eiffel, a nova linguagem permite o uso de argumentos covariantes na redefinição de métodos em subclasses, nunca descurando o uso do modificador **covariant**, que identifica a possibilidade de covariância nos argumentos de um método. A introdução do tipo genérico **This** visa explorar ao máximo o potencial do mecanismo de covariância, quer pelo seu uso nos argumentos de métodos, onde se apresenta como uma solução para os métodos binários, quer pela possibilidade de ocorrer no resultado das funções e na definição de variáveis locais e de instância. O caminho seguido na implementação deste mecanismo foi o do uso de tipificação dinâmica de forma substancial, sendo que também se fazem diversas validações estáticas de tipos, na medida do possível. Ao longo da descrição do trabalho feito procurou-se mostrar através de mini-exemplos a utilidade que este mecanismo pode ter na programação.

O trabalho consistiu no desenvolvimento de um tradutor, implementado recorrendo à ferramenta SableCC, que recebe à entrada um ficheiro de um programa em JavaCO e gera um ficheiro em Java, resultante da tradução. Após a realização deste trabalho, pode concluir-se que foram atingidos os objectivos propostos. Nomeadamente, manteve-se a compatibilidade com os programas em Java puro, foi mantida a invariância dos tipos dos argumentos dos métodos à saída do tradutor, evitou-se a repetição das validações de tipos feitas pelo compilador de Java e definiram-se com sucesso os erros a emitir relativamente à covariância.

É de salientar que a maior parte do trabalho, em termos de volume e esforço, foi dedicado a tentar compreender a essência dos mecanismos a implementar, e posterior expressão das

ideias obtidas, usando o tradutor. O facto do Java ser uma linguagem muito grande e com enorme riqueza sintáctica tornou o trabalho bastante exigente, levando-nos a manter um cuidado constante para evitar o esquecimento de qualquer caso particular que tivesse de algum modo relacionado com os mecanismos que estavam a ser definidos.

A nível pessoal, este trabalho fez com que passasse a olhar para as linguagens de programação de outra maneira. Ajudou-me a perceber o impacto que têm as particularidades das linguagens de programação, na forma como todas essas particularidades contribuem para um todo coerente. Por exemplo, foi interessante ver que o mecanismo único que foi introduzido na linguagem JavaCO, e que parece limitado, está cheio de ramificações e tem mais detalhes a ter em conta do que aqueles que inicialmente previa.

Foi igualmente enriquecedora a experiência de exprimir o sistema de tipos da nova linguagem através da linguagem Java, o que permitiu aprofundar os conhecimentos sobre o próprio Java.

É também de realçar a curiosidade deste tradutor que nos permite escrever código que, indirectamente, vai gerar código para fazer as validações dinâmicas de tipo por nós.

Por fim, resta frisar que a realização deste trabalho permitiu aplicar os conhecimentos adquiridos ao longo tanto da Licenciatura como do Mestrado em Engenharia Informática, em particular relativamente às linguagens de programação.

Bibliografia

[Cardelli 97] – Cardelli, L.: "*Type Systems*". Handbook of Computer Science and Engineering, Chapter 103. CRC Press, 1997

[Copeland 07] – T. Copeland: "*Generating Parsers with JavaCC", 2007*

[E. Gamma et al. 94] – E. Gamma, R. Helm, R. Johnson, J. Vlissides: "*Design Patterns: Elements of Reusable Object-Oriented Software",* 1994

[Eiffel web 1] – Invitation to Eiffel – Disponível em: http://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-00.html

[Eiffel web 2] – An Eiffel Tutorial **–** Disponível em: http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial-00.html

[Eiffel web 3] – The Eiffel Method and Language – Eiffel Documentation – Disponível em: http://docs.eiffel.com/book/method/method

[F. Miller et al. 10] – F. Miller, A. Vandome, J. McBrewster: "*Covariance and Contravariance (computer Science): Type System, Programming Language, Subtype Polymorphism, Category Theory, Covariance and Contravariance of Vectors, Class Hierarchy, Object-Oriented Programming, Liskov Substituiton Principle"*, 2010

[Gosling, McGilton 96] – J. Gosling, Henry McGilton: *"The Java Language Environment"*, 1996

[JavaCC web 1] – JavaCC tutorial - http://www.engr.mun.ca/~theo/JavaCC-Tutorial

[JavaCC web 2] – JavaCC FAQ - http://www.engr.mun.ca/~theo/JavaCC-FAQ

[João 04] – João, M.; Frade, G.: "*Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi*", 2004

[K. B. Bruce et al. 03] – K. Bruce, A. Schuett and R. van Gent, "PolyTOIL: A type-safe, polymorphic object-oriented language," *ACM Trans. Prog. Langs. and Sys.*, 225-290, 2003

[Gagnon 98] – Gagnon, E: "*SableCC, An Object-Oriented Compiler Framework*", Master's thesis, McGill University, Montreal, 1998 – Disponível em: http://sablecc.sourceforge.net/downloads/thesis.pdf

[Meyer 05] – Meyer, B.: "*Attached Types and their Application to Three Open Problems of Object-Oriented Programming*" ,in ECOOP 2005, Springer Verlag, pages 1-32, 2005

[Meyer 06] – Meyer, B.: "*Eiffel: The Language, third edition"*. Draft 5.10. Santa Barbara, California. Agosto 2006

[Meyer 97] – Meyer, B.: "*Object-Oriented Software Construction, 2nd Edition*", Prentice Hall, 1997

[Nystromet et al. 03] – Nystrom, N.; Clarkson, M.; Myers, A.: "*An Extensible Compiler Framework for Java*", Proceedings of the 12th International Conference on Compiler Construction, Warsaw, Poland, April 2003

[Omohundro, Lim 92] – S. Omohundro and C. Lim, "*The Sather Language and Libraries*", International Computer Science Institute, 1992

[Pierce 02] – Pierce, B.: "*Types and Programming Languages"*. MIT Press. Cambridge, Massachusetts, London, England, 2002

[Polyglot web] – Polyglot web site – Disponível em: http://www.cs.cornell.edu/projects/polyglot/

[SableCC web 1] – SableCC web site - http://sablecc.org/

[SableCC web 2] – Beginner's Guide to Using SableCC with Eclipse web site – Disponível em: http://www.comp.nus.edu.sg/~sethhetu/rooms/Tutorials/EclipseAndSableCC.html

[Sather web] – Sather Web site – Disponível em: http://www.icsi.berkeley.edu/~sather/

[Schaffert et al. 86] – C. Schaffert, T. Cooper, B. Bullis, M. Killian and C. Wilpolt, "An Introduction to Trellis/Owl," ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 9-16, 1986

[Shaker, Norvell 04] – Shaker, P.; Norvell, T.: *The Java Parsing System*, 2004

ANEXO I – código ilustrativo de alguns aspectos da tradução

- Retirar o modificador **covariant**:

JavaCO

```
public covariant void comer(Alimento al)
```

TRADUTOR

Java

```
public void comer(Alimento al)
```

```
public void outACovariantModifier(ACovariantModifier node){
                node.replaceBy(new ACovariantModifier());
}
```

O nó original da AST do programa de entrada relativo ao modificador **covariant** é substituído por um novo nó, do mesmo tipo *ACovariantModifier*, não contendo no entanto qualquer texto (neste caso “covariant”).

- Troca do tipo de cada argumento covariante pelo melhor tipo encontrado pelo algoritmo BestTypes e adição de carácter *underscore*:

JavaCO

```
class Vaca extends Animal {
  public covariant void m (Vaca v) { }
}
```

TRADUTOR

Java

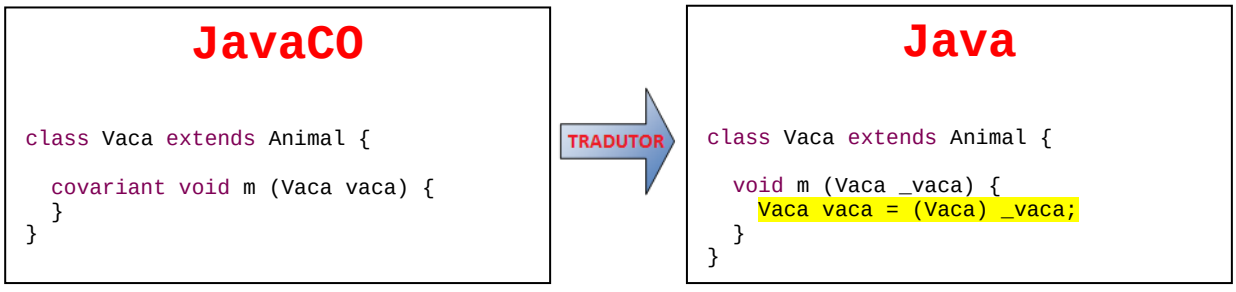
```
class Vaca extends Animal {
  public void m (Animal _v) { }
}
```

```
public void outAReferenceFormalParameter(AReferenceFormalParameter node) {

       if (isInsideMethodDeclaration) {

               Node currentMethod = stackMethods.peek();

               if (Globals.methods.get(currentMethod).isCovariant()) {

                  if (Globals.methods.get(currentMethod)
                       .getCovariantArguments().contains(node)) {

                       int indexForTestingIfIsGenericType = Globals.methods.get
                           (currentMethod).getCovariantArguments().indexOf(node);
                       String s1 = Globals.methods.get(currentMethod)
                           .getCovariantArgumentsToCheckLaterIfIsGenericType()
                           .get(indexForTestingIfIsGenericType);

                       boolean isTypeParameterClass = false;
                       boolean typeArgumentsAreAllParametricTypes = false;

                       if ( Globals.classesAndInterfaces.get(s1) != null ) {
                         if ( Globals.classesAndInterfaces.get(s1).isTypeParameterClass() ) {
                            // do nothing
                            isTypeParameterClass = true;
                         }
                       }

                       if ( ! isTypeParameterClass ) {  // yet
                         if (s1.contains("<")){
                           LinkedList<String> listaaa = Globals.separateArgumentWithParametricTypes(s1);
                           boolean allParametrictypes =
                             Globals.checkIfAllTypeArgumentsAreParametricTypes(listaaa);

                           typeArgumentsAreAllParametricTypes = allParametrictypes;

                           if ( typeArgumentsAreAllParametricTypes )

                            node.setIdentifier2(new TIdentifier("_" + node.getIdentifier2().getText()) );
                         }
                       }

                       if ( ! isTypeParameterClass && ! typeArgumentsAreAllParametricTypes ) {

                          int index = Globals.methods.get(currentMethod).getMethodHeader()
                               .getArgumentNodes().indexOf(node);

                          LinkedList<LinkedList<String>> aLista = Globals.methods.get(currentMethod)
                               .getMethodHeader().getBestTypes().getParents().get(index);

                          String typeToReplace = Globals.bestTypeToReplace(
                               Globals.intersectionOfAllLists( aLista ) );
                          typeToReplace = typeToReplace.substring(1, typeToReplace.length()-1);

                          if ( typeToReplace.contains("#") ) {

                           int ind = typeToReplace.lastIndexOf("#");
                           typeToReplace = typeToReplace.substring(ind, typeToReplace.length() );

                           boolean done = false;
```

```
        while (!done){
              char c = typeToReplace.charAt(0);
              typeToReplace = typeToReplace.substring(1, typeToReplace.length());
                if (c == '.')
                   done = true;
         }
       }


       if ( Globals.classesAndInterfaces.get("." + typeToReplace + ".") != null )
         if ( Globals.classesAndInterfaces.get("." + typeToReplace + ".").isTypeParameterClass() ){

              int indexOfLastDot = typeToReplace.lastIndexOf(".");
              typeToReplace = typeToReplace.substring(indexOfLastDot+1, typeToReplace.length());
         }

         if ( node.getDims1() != null ) {
           Iterator<PDim> it = node.getDims1().iterator();
           while(it.hasNext())
             typeToReplace += it.next().toString().trim();
         }

       node.getIdentifier1().setText( typeToReplace );
       node.setAdditionalIdentifiers(new LinkedList<PAdditionalIdentifier>());
       node.setDims1(new LinkedList<PDim>());
       node.setDims2(new LinkedList<PDim>());
       node.setTypeArguments(null);
       node.setTypeComponents(new LinkedList<PTypeComponent>());
       node.setIdentifier2( new TIdentifier("_" + node.getIdentifier2().getText()) );

       }
     }
   }
 }
}
```

É aqui apresentado o código que trata de fazer a alteração dos argumentos dos métodos covariantes do programa de JavaCO para Java puro: o tipo de cada argumento covariante é alterado para o melhor tipo descoberto pelo algoritmo *BestTypes*, e ao identificador do argumento é adicionado um carácter *underscore*.

- Criação de variável auxiliar para validação dinâmica do tipo de um argumento covariante:



```
private AVariableDeclarationBlockStatement createAVariableDeclaration(AReferenceFormalParameter argument,
                                                    String realNameOfThis, boolean isThisVariable) {

        if( realNameOfThis != null ) {

          // caso em que o tipo é um parâmetro de tipo
          if (  Globals.classesAndInterfaces.get( "." + realNameOfThis + "." ) != null )
            if ( Globals.classesAndInterfaces.get( "." + realNameOfThis + "." ).hasTypeParameters() ) {

                realNameOfThis = Globals.addFullTypeParametersNamesToClassName("."+ realNameOfThis +"." );
            }
          }


        // Parte partilhada, vai ser diferente caso se esteja ou não a tratar de uma variável This
        AUnaryUnaryExpression unaryUnaryExpr;


/*************************************************************************************************************
*************************************************************************************************************
*************************************************************************************************************/


        // Se a variável não é This
        if ( ! isThisVariable ) {

          AReferenceIdentifierCastExpression refIdCastExpr = new AReferenceIdentifierCastExpression();
          String identifier2 = argument.getIdentifier2().toString().trim();
          int indexOfId2 = argument.toString().lastIndexOf(identifier2);
          String fullIndentifier1 = argument.toString().substring(0, indexOfId2);

          refIdCastExpr.setIdentifier1(new TIdentifier(
                Globals.deleteTypeArgumentsFromTypeToCast(fullIndentifier1) ));

          refIdCastExpr.setIdentifier2(new TIdentifier("_" + argument.getIdentifier2().getText()));

          refIdCastExpr.setLPar(new TLPar());
          refIdCastExpr.setRPar(new TRPar());

          ACastUnaryExpressionNotPlusMinus castUnaryExprNotPlusMinus = new
                ACastUnaryExpressionNotPlusMinus(refIdCastExpr);

          // parte partilhada, neste caso não se está a tratar variável This
          unaryUnaryExpr = new AUnaryUnaryExpression(castUnaryExprNotPlusMinus);
        }

/*************************************************************************************************************
*************************************************************************************************************
*************************************************************************************************************/

        else { // estamos a tratar uma variável This

          //this
          AThisPrimaryNoNewArray thisPrimNoNewArr = new AThisPrimaryNoNewArray( new TThis() );
          ANoArrayPrimary noArrayPrimary1 = new ANoArrayPrimary( thisPrimNoNewArr );

          APrimaryMethodInvocation primaryMethodInvocation = new APrimaryMethodInvocation();
          primaryMethodInvocation.setIdentifier( new TIdentifier( "getClass" ) ); //  -> getClass
          primaryMethodInvocation.setDot( new TDot() );
          primaryMethodInvocation.setLPar( new TLPar() );
          primaryMethodInvocation.setRPar( new TRPar() );
          primaryMethodInvocation.setPrimary( noArrayPrimary1 );  //      -> this
```

```
          AMethodPrimaryNoNewArray methPrimNoNewArray = new AMethodPrimaryNoNewArray(
                primaryMethodInvocation );
          ANoArrayPrimary noArrayPrimary2 = new ANoArrayPrimary( methPrimNoNewArray );
          APrimaryMethodInvocation primaryMethodInvocation2 = new APrimaryMethodInvocation();

          AOneIdArgumentList oneIdArgumentList = new AOneIdArgumentList();
          oneIdArgumentList.setIdentifier( new TIdentifier("_" + argument.getIdentifier2().getText()) );

          primaryMethodInvocation2.setArgumentList(oneIdArgumentList);
          primaryMethodInvocation2.setDot( new TDot() );
          primaryMethodInvocation2.setIdentifier( new TIdentifier("cast") );
          primaryMethodInvocation2.setLPar( new TLPar() );
          primaryMethodInvocation2.setRPar( new TRPar() );
          primaryMethodInvocation2.setPrimary(noArrayPrimary2);

          AMethodPrimaryNoNewArray methodPrimaryNoNewArray = new AMethodPrimaryNoNewArray(
                primaryMethodInvocation2 );
          ANoArrayPrimary noArrayPrimary3 = new ANoArrayPrimary( methodPrimaryNoNewArray );

          APrimaryPostfixExpression primaryPostfixExpression = new APrimaryPostfixExpression(
                noArrayPrimary3 );
          APostfixUnaryExpressionNotPlusMinus postfixUnaryExpressionNotPlusMinus = new
                APostfixUnaryExpressionNotPlusMinus( primaryPostfixExpression );

          // parte partilhada, aqui estamos a tratar uma variável This
          unaryUnaryExpr = new AUnaryUnaryExpression( postfixUnaryExpressionNotPlusMinus );

       }


/*************************************************************************************************************
**************************************************************************************************************
**************************************************************************************************************/

       ASimpleMultiplicativeExpression simpleMultExpr = new
              ASimpleMultiplicativeExpression(unaryUnaryExpr);
       ASimpleAdditiveExpression simpleAddExpr = new ASimpleAdditiveExpression(simpleMultExpr);
       ASimpleShiftExpression simpleShiftExpr = new ASimpleShiftExpression(simpleAddExpr);
       ASimpleRelationalExpression simpleRelExpr = new ASimpleRelationalExpression(simpleShiftExpr);
       ASimpleEqualityExpression simpleEqExpr = new ASimpleEqualityExpression(simpleRelExpr);
       ASimpleAndExpression simpleAndExpr = new ASimpleAndExpression(simpleEqExpr);
       ASimpleExclusiveOrExpression simpleExclOrExpr = new ASimpleExclusiveOrExpression(simpleAndExpr);
       ASimpleInclusiveOrExpression simpleInclOrExpr = new
              ASimpleInclusiveOrExpression(simpleExclOrExpr);
       ASimpleConditionalAndExpression simpleCondAndExpr = new
              ASimpleConditionalAndExpression(simpleInclOrExpr);
       ASimpleConditionalOrExpression simpleCondOrExpr = new
              ASimpleConditionalOrExpression(simpleCondAndExpr);
       ASimpleConditionalExpression simpleCondExpr = new ASimpleConditionalExpression(simpleCondOrExpr);
       AConditionalAssignmentExpression condAssignExpr = new
              AConditionalAssignmentExpression(simpleCondExpr);
       AExpression expr = new AExpression(condAssignExpr);
       AExpressionVariableInitializer exprVarInit = new AExpressionVariableInitializer(expr);

       AInitializerVariableDeclarator initVarDecl = new AInitializerVariableDeclarator();
       initVarDecl.setAssign(new TAssign());
       initVarDecl.setIdentifier(new TIdentifier(argument.getIdentifier2().getText()));
       initVarDecl.setVariableInitializer(exprVarInit);

       AOneVariableDeclarators oneVarDecl = new AOneVariableDeclarators(initVarDecl);
       AReferenceLocalVariableDeclaration refLocalVarDecl = new AReferenceLocalVariableDeclaration();
```

```
     if(argument.getIdentifier1().getText().equals("This")) {

        String aux  = Globals.removeDotsFromTypeNameWithTypeArguments(realNameOfThis);
        refLocalVarDecl.setIdentifier( new TIdentifier( aux ) );
     }

     else {

        String identifier2 = argument.getIdentifier2().toString().trim();
        int indexOfId2 = argument.toString().lastIndexOf(identifier2);
        String fullIndentifier1 = argument.toString().substring(0, indexOfId2);
        refLocalVarDecl.setIdentifier(new TIdentifier( fullIndentifier1 ));
     }

     refLocalVarDecl.setVariableDeclarators(oneVarDecl);
     ALocalVariableDeclarationStatement localVarDeclStat = new ALocalVariableDeclarationStatement();
     localVarDeclStat.setLocalVariableDeclaration(refLocalVarDecl);
     localVarDeclStat.setSemi(new TSemi());

     AVariableDeclarationBlockStatement stat = new AVariableDeclarationBlockStatement(localVarDeclStat);

     return stat;

}
```

O código em cima transcrito é utilizado para a criação dos nós correspondentes às variáveis auxiliares para a validação dinâmica dos argumentos covariantes. Os nós gerados, do tipo *AVariableDeclarationBlockStatement*, serão acrescentados à Árvore de Sintaxe Abstracta “à cabeça” do corpo dos métodos respectivos.